Confidentiality Properties and the B Method

Onunkun, T J

Awarding institution:
King's College London

Confidentiality Properties and The B Method

presented to

The Software Modelling and Applied Logic Group

at the

Department of Informatics
King's College London

In Partial Fulfillment of the Requirements for
a PhD

by

Temitope Jos Onunkun

September 1, 2012

Abstract

Programs in the presence of nondeterminism or underspecification may mask the presence of insecure information flow between variables. This may result in the refinement paradox when such programs are refined to a deterministic implementation. Hence nondeterministic programs that satisfy possibilistic security properties like Generalised Noninterference (GNI) may, on refinement, fail corresponding deterministic security properties such as Noninterference (NI).

We propose in this thesis an automatable information flow analysis framework to capture information flow between variables and flag flows that breach information flow policies defined as a multi-level secure lattice-based system. We separate the problem of satisfaction of the refinement relation from the problem of preservation of security properties of interest at every refinement step, and focus on the latter problem.

We formalise our core analysis on *standalone* B Machines, develop the proof obligations of the framework, and introduce security conditions that must be satisfied to guarantee secure information flow between the variables within a single B machine (Chapter 3). We show that our analysis is more robust than standard *flow-insensitive* security type systems like the one developed by Volpano, Smith, and Irvine [76], since our analysis is *flow-sensitive*, i.e., responsive to information flow. For example, our framework correctly analyses a program whose overall flow is secure as *secure*, even when some of its subprograms may be insecure, whereas [76] will erroneously classify such programs as *insecure*, a problem commonly termed *false negative*. We also show the correctness of our framework in Chapter 3.

A natural sequel to our core information flow analysis of standalone B Machines is an extension of the framework to analyse *structured* B Machines, i.e., information flow arising from the use of B structuring mechanisms such as **SEES**, **INCLUDES**, etc (Chapter 4).

The third major part of the thesis (Chapter 5) involves the analysis of information flow between the variables in a hypothetical case study using the C++ implementation of the information flow analyser formalised in the preceding chapters. We also discuss our intuitions on future extensions.

Acknowledgements

The aphorism[1] "*Standing on the shoulders of giants*", allegedly first recorded in the twelfth century and attributed to Bernard of Chartres [104], [77], [78]; reportedly restated by Isaac Newton (seventeenth century) is certainly true in my case. I owe much to the many giants who have offered me their shoulders to stand on ...

To David Clark, my first supervisor, for his guidance, patience, and assistance in honing the mathematical skills needed to formalise my ideas, and for faithfully providing frank critique of my work even after moving from King's College London to University College London.

To Sebastian Hunt and Pasquale Malacaria, who probably did not realise how useful they were to me during the early years of my research through informal discussions they had with David and I;

To Juan Bicarregui whose insight into confidentiality properties and the B method helped me formulate the path I later followed in the research leading up to this thesis;

To Kevin Lano and Iman Poernomo, my new supervisors, who have been very helpful, especially with the B Method and programming concepts arm of this thesis. From Abrial to Zdancewic, I am indebted to those giants too numerous to list here who have gone ahead, and on whose works I here build.

To Mark Harman and all other members of the Centre for Research on Evolution, Search and Testing (CREST) now based at University College London for providing a vibrant environment where research students and experienced researchers could present and discuss their ideas.

To Michael Luck for the instructive workshop on "How to finish a PhD and have a successful Viva" (see [88]) and his ongoing support of PhD students at Kings College London.

I acknowledge the administrative, systems and academic members of staff at the Department of Informatics, Kings College London, for their friendly support and genuine efforts in providing a conducive research environment for us research students. Ditto the Kings Learning and Teaching

[1] A short clever saying expressing a general truth or principle; an adage.

Institute staff for the excellent training and support provided in Academic Practice; and thanks to the staff at Maugham Library for their assistance from time to time in tracking down reference material. I very much thank my colleagues Adam Wymer, Kelly Androutsopoulos, Arlene Ong, Huzam Al-Subaie, Chunyan Mu, Khalid Alzarouni, Assel Akzhalova, and Arpit Gupta for constructive discussions, reviews and their general support. Yes, for the unwinding fun times we had together too. To the *unnamed ones* who have supported me at some point, accept please my sincere thanks. And, of course,

To my family and friends I owe a debt of gratitude for their support and undying faith in me. Most assuredly I owe a debt of gratitude to my late little brother, Dr Olumuyiwa Onunkun, former Medical Director of M&V Hospital (now Muyiwa Onunkun Memorial Hospital) Ore Nigeria, whose encouraging words never to give up my dream, even while on the sick bed from which he never recovered, continued to ring in my ears through the sometimes dark days of the work leading up to this thesis.

Temitope Jos Onunkun (aka TJ) Kings College London.

September 1, 2012

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

It used to be the case that many software professionals in the industry focused on the functional requirements of software systems almost to the exclusion of nonfunctional requirements like confidentiality. Hence, confidentiality, and computer security mechanisms in general, were sought as add-on components to already developed software systems. In the last decade or so, however, a growing number of software professionals have started to incorporate some form of confidentiality properties, usually security-typing frameworks, into the development process. Examples include JIF (an extension of Java that supports information flow and access control), SIF (Servlet Information Flow, that does for servlets what JIF does for java objects). We present in this thesis the theoretic foundations on which our research into *Confidentiality Properties and the B Method* is based, and a practical framework for building confidentiality respecting software systems. Using the B Method, we illustrate with a case study how existing knowledge in the area of confidentiality properties can be harnessed in the specification and development of software systems in order to guarantee that these properties are preserved at every step of the development process.

1.1 Setting

A statement made by Dorothy Denning in May 1976 is as valid now in 2011 as it was back then. The author wrote in [50] that '*The primary difficulty with guaranteeing security lies in detecting (and monitoring) all flow*

causing operations.' Hence, a major research problem we intend to address in this thesis is: '*the analysis of information flow to the end of monitoring and detecting possible insecure flows within B developments based on defined confidentiality policies*'. To simplify this problem, we propose a decoupling of the problem of preservation of confidentiality properties from the problem of preservation of the refinement relation. Thus, rather than constrain the classical refinement relation, thereby overhauling the foundations of this traditional software development framework, we extend the software development process with an additional step which analyses specifications and their refinements for insecure flows of information. This 'plug-in' approach means that if a refinement of a system satisfies the classical refinement relation, and our framework analyses the system and adjudges it to have secure flows of information, then we can conclude that such a system is, with respect to the defined confidentiality policy, a secure refinement of the original specification. This notion is discussed in more detail in Section 3.2.

It is worth pointing out that our approach here, in principle, is akin to the notion of Aspect-Oriented Software Development (AOSD), an engineering approach whereby the secondary requirements of software systems are separated from the primary business logic concerns and built as separate modules to be integrated with the primary systems. At the source code level, this engineering paradigm is referred to as Aspect-Oriented Programming (AOP) [83]. For example, in the development of a system for storing and manipulating employee records, *access* and *change* requirements may affect several parts of the system. So rather than entangle these requirements in different parts of the source code thereby making traceability and future modification more difficult, in AOP such concerns (termed cross-cutting concerns) are separated from the primary system requirement of bookkeeping and indexing of the employee records. However, because AOP is employed only at the level of source code, whereas the scope of the B Method, which we employ in our work, cuts across all levels of the software development process, we did not employ AOP in our research.

A core part of our work involves an extension of the flow logic analysis formalism introduced by Clark et al in [36] to the B Method. Since our framework is built on the Generalised Substitution Language (GSL) of the

B Method, we illustrate our work with examples, for the most part, written using the B Abstract Machine Notation (AMN). Although other formal specification languages were considered, our reasons for choosing the B Method are discussed in Section 1.2. First, we precís information flow security, showing the limitations of current general approach towards attaining it.

Information flow theory deals with the transfer of information from one point (*source*) to another (*destination*) via some media or virtual communication channel. Given the pervasive nature of computer-based information today, the need to protect electronic data, and associated resources from accidental and/or malicious modification, leakage and damage is more important than ever. This is the primary objective of *computer security*. The US National Information Systems Security (INFOSEC) Glossary 2000 defines computer security as "*measures and controls that ensure confidentiality, integrity, and availability of Information system assets including hardware, software, firmware, and information being processed, stored, and communicated*" [112].

Information flow security is an aspect of computer security requiring that secret information is not inadvertently leaked to unauthorised observers, often referred to as adversaries. Generally, we say information flows from an "entity"[1] E_1 to another "entity" E_2 if the value of E_2 is modified, or influenced in some way by E_1 (or by an entity derived from E_1). Information flow could be represented graphically as shown in Figure 1.1, where I_1 denotes information flow from source s through a communication channel, c, and I_2 denotes information received at destination, s'. Ideally, we desire that the adversary's knowledge of any secret in I_1 is not increased by his legitimate observations of I_2 at s' in all runs of the system.

Information flow security is often treated as a *confidentiality property*[2], and this could be described informally as:

- The prevention of unauthorised access to secret information explicitly,

[1] We use the term 'entity' to describe all logically identifiable objects such as variables, components, files, stored values, segments, etc. in a software system

[2] Confidentiality properties are also referred to as secrecy (when associated with data or information) or privacy properties (when described as a user requirement).

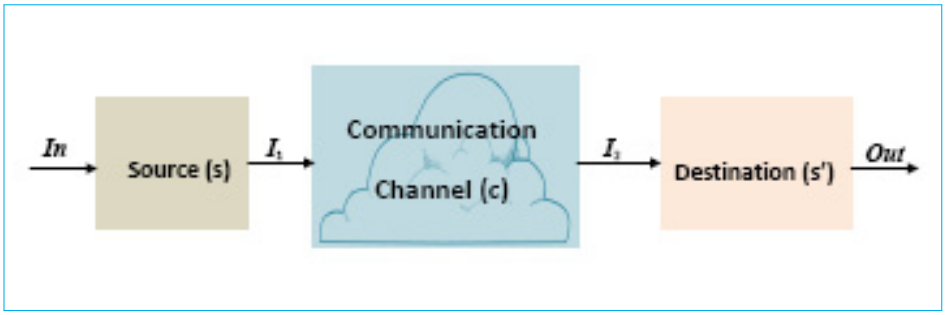


Figure 1.1: Information flow

and

- The prevention of unauthorised persons from *learning* or deducing transmitted confidential information by covertly observing the behavioural patterns of the system, e.g., timing behaviour, input and output behaviour, resource usage, and/or termination or otherwise of a process based on information publicly available to them.

The INFOSEC Glossary [112] defines confidentiality as: "*assurance that information is not disclosed to unauthorized persons, processes, or devices*". Thus we need to be able to guarantee that confidential information remains confidential from one end of the system to the other (i.e., end-to-end security). Confidentiality therefore requires that secret information does not flow into output receptacles visible to public (or unauthorised) observers. The constraints or restrictions imposed on information flow to ensure the confidentiality of sensitive (or secret) information within software systems are termed *confidentiality properties* [19].

However, many software systems in use today depend on what Gary McGraw [93], [94] termed a "*penetrate and patch*" (p&p) approach to security whereby piecemeal solutions (patches) are provided to fix "known" vulnerabilities in software, much like patching leaks in a vessel. This inevitably weakens the structure of the original program thereby resulting sometimes in system instability. Ghosh et al [59] concurs with [93] who notes that to avoid the problems sometimes caused by these patches, many users are often reluctant to apply the patches, as long as they have a '*working*' system. Furthermore, a software system is only as secure as its last known vulnera-

bility, hence, there is an ever-present danger of *unknown* vulnerabilities in software systems. Often, a number of vulnerabilities go unnoticed for a long time because adversaries exploiting such vulnerabilities will obviously not be inclined to report them to the developers.

It must be noted that many vulnerabilities, for example language-based flaws like buffer overflows, can be prevented by the use of strict formal development methods like the B Method. Strict formalisms, however, cannot deal with information flow related '*security holes*' in software systems. Consequently, rather than spinning round and round in the '*penetrate and patch*' carousel, which at best only deals with symptoms, we believe it is much better to deal with the underlying problem by thinking about ways of building end-to-end confidentiality respecting software systems from inception step-by-step through to implementation.

In the following section, we present the reasons why we consider work on the preservation of confidentiality properties in a formal software development environment such as the B Method a necessary and beneficial endeavour.

1.2 Motivation

Article 8 of the European Convention on Human Rights [46], the Data Protection Act 1998 [47], [13], the Digital Millennium Copyright Act of 1998 (US) [52], Copyright laws, Design Rights, Intellectual Property Rights [43]

> ... all these constitute "constraints" on governments, individuals and organisations to ensure protection of the confidential information they hold about other individuals and/or organisations.

As the world is becoming increasingly dependent on computers, more and more data are becoming digitised, stored on computers, and manipulated in ever more ingenious ways, hence the need to seek better ways of preserving the confidentiality of such information from end to end in a software system cannot be overemphasised. That the protection and preservation of personal information held on individuals by governments and organisations

is important can be seen from people's indignation and government officials' embarrassment at the recent spate of leaks of such information by *wikileaks*.

Another motivation for our work is the need for a formal engineering approach to specification and refinement of software systems that allows practitioners to model provably correct and secure programs before actual development takes place, thereby reducing undue reliance on testing and validation after program construction. This will make the software development process much more like other engineering fields where such an approach is the norm. For example, rather than build a bridge and then test it's load resistance, a civil engineer will produce formal drawings and models which are then tested as a simulation of the real bridge. Researchers like Burgess [35], Ghosh et al [59], and McGraw [93] discussed the importance of such an engineering approach to software development. This approach makes sense both technically and economically. It is more cost effective to spot and fix secure information leaks early on in the development process than to discover these later-on during the testing or deployment phase, for example, in the traditional waterfall software development model. Gilb and Graham [60], among others, claim it could be up to 30% cheaper in time and cost to inspect and fix software problems (confidentiality, in our context) earlier than later [45]. The information flow analysis framework introduced in this thesis provides such a *formal engineering approach* to the specification and refinement of confidentiality sensitive software systems, with particular focus on B developments.

One reason we decided to use the B Method is because of the need for solutions to be simple enough for it to be readily adopted by practitioners (many of whom lack advanced mathematical training) in the industry, without completely overhauling well-established development frameworks. Banach and Poppleton [17], in the context on *software retrenchment* [3], stressed the need not to impose "an alien development discipline on an already well-established engineering milieu". They argued that "engineers with an established track record of successful development are seldom sympathetic to the suggestion

[3]Retrenchment (introduced by Banach and Poppleton) simply is the opposite of refinement; "loosely-speaking the strengthening of the precondition and the weakening of the postcondition" [18], [16], [137]

that all their familiar working practices ... be abandoned in favour of a way of working forced implicitly by ..."[17] some *formalism*. The B Method is a well-known formal specification method based on set theory, and it is familiar to many industry practitioners and academics. That many researchers are increasingly conscious of the need for solutions to be simple is pointed out by Finklestein and Kramer in [55], and Denning in [50]. Benjamin Aziz in [14], [138] stressed the need for integration of formal methods into industrial solutions, as exemplified by the 'EU FP7 project CONSEQUENCE', which delivered "*a data-centric information protection framework based on data-sharing agreements*" [136]. Hence, defining our information flow analysis framework on B GSL semantics, rather than on a toy imperative language as is commonly done in the literature, and developing an analysis tool that can be integrated into the formal development process of B machines fills a need in the industry. Another motivation for using the B Method is the avaialability of well-known industrial tools such as the B Toolkit, ProB, Atelier B, etc, for simulating and validating B Machines.

Our motivation for *automating* the information flow analysis framework lies in the fact that it minimises the risk of human error and reduces labour, training and time investment in manually analysing information flow at every step of the development process. This we believe is more cost-effective and allows for consistency of results, as developers only have to learn how to use the information flow analysis tool to check for insecure flows in their development rather than learn the rigorous formal analysis framework.

We employ a lattice structure to formalise security policies that map program variables into security classes, following Denning's approach in [50]. Lattices are special types of sets, so it is only natural to use a set-based formal method like the B Method for our analysis and supporting examples. This, of course, is not to say that specification and/or programming languages not based on set theory cannot be used. In summary, the simplicity of the B Method, its expressive power and compatibility with security lattices, its use of familiar set-theoretic approach, availability of commercial tool support and increasing popularity in industry makes the method our formal specification language of choice [68], [123].

Consequently, we summarise below the motivation for our work, based on the belief that practitioners need to build confidentiality properties into software systems from an early stage of the software development life cycle and to ensure that such properties are preserved through refinement(s) to implementation:

- The storage and manipulation of large amounts of sensitive confidential information on computer systems;
- The interconnection of countless disparate systems in an ever-expanding maze of internetworked systems through the internet, some of which cannot be trusted;
- Readily exploitable confidentiality related flaws in many existing software systems, which could cause significant loss of reputation, competitive advantage, customer revenue ... and even life! [13];
- Present "penetrate and patch" approach cannot deal with the threat posed by adversaries, identity thieves, terrorists, the '*wikileaks*' of this world, etc;
- The potential effect of human error when information flow is analysed manually;
- The need to satisfy government laws and regulations such as the Data Protection Act, Freedom of Information Act, etc;
- The need to develop simple, cost-effective, yet powerful and secure solutions that can be readily assimilated into industry and academia.

Having discussed the reasons we have chosen to carry out this research work, we now present the aims and objectives of the thesis in Section 1.3 below.

1.3 Aims and Objectives of Thesis

The aim of this thesis is to extend existing theories on the subject of confidentiality properties to formulate an automatable information flow analysis framework to enable developers using the B Method to assure end-to-end

secure information flow within their developments.

To accomplish our stated aim, we identify the following objectives:

- A study and review of relevant possibilistic security properties in the literature;
- A study and review of the notion of programs, specifications and refinements used in the literature, and limitations of the classical refinement relation;
- A study and review of the B Method, with examples to show that *formalism* does not automatically translate to security;
- Information flow analysis of systems developed using the Generalised Substitution Language of B Machines;
- Introduction of security conditions for a multi-level secure system, using a lattice structure;
- A B development case study *automatically* analysed for secure information flow using our C++ Implementation of the information flow analysis framework introduced in this thesis;

We present in Section 1.4 the problem addressed in this thesis and the research methodology used to deal with it.

1.4 Research Methodology and Plan

We present in this section the research method employed in our work as well as our research plan, which sets out the road map for accomplishing the aims and objectives presented in Section 1.3.

1.4.1 Research Methodology

The problem we address in this thesis is how to assure end-to-end secure information flow within systems developed using the B Method. The research methodologies we employ include reviews, action research, and case-studies.

We investigate possibilistic confidentiality properties in the literature; existing notions of programs, specifications and refinements; and we present an introduction to the B Method. We present a detailed discussion of this part of our research in Chapter 2 of the thesis. We also employ *Action Research* [9] methodology, which consists of an iterative spiral of steps involving planning, action, and a review of completed actions with a view to improving the process. We follow our review of existing work in the literature with a discussion of the pros and cons of existing approaches to the preservation of confidentiality properties through refinement (Section 2.3.5.5). We then break the problem of preservation of confidentiality properties through refinement into two distinct concerns that must both be guaranteed from one development step to another, namely: the preservation of the *refinement relation*, and the preservation of the *confidentiality properties* of interest. We discuss this in Chapter 3.

We adopt the notion of refinement in the B Method, which corresponds to set inclusion of the behaviours of the refined system in the behaviours manifest by the original specification, hence we concentrate only on the second strand of the problem stated above, i.e., the preservation of confidentiality properties of interest. To do this, we develop a flow-logic based analysis of information flow between parts of the system state and introduce security conditions that, if satisfied, guarantee that such a system satisfies the confidentiality property of interest. We apply these conditions first to Generalised Noninterference and Noninterference and *review* the results with a view to optimising the analysis (Sections 3.5 - 3.8).

Our action research method corresponds to Stephen Corey's (1953) view of "Action Research" as "the process by which practitioners attempt to study their problems scientifically in order to guide, correct, and evaluate their decisions and actions"[40] in a disciplined self-reflective fashion. Earlier, Kurt Lewin (1947) defined the same approach as "a three-step spiral process of (1) planning, which involves reconnaissance; (2) taking actions; and (3) fact-finding about the results of the action"[40]. Hence we aim to use the results of our reflective research process to seek an optimal information flow analysis framework that will be robust enough to handle all possibilistic definitions of security in the literature. This leads to our discussion of

analysis of information flow using monotone frameworks in Section 3.8.

To demonstrate how our analysis framework can be utilised in the industry, we employ the *case-study* research methodology. Using the B Method, we introduce a system development methodology that enables us to analyse the flow of information between the variables within standalone B machines, refinements and implementations, as well as information flow between variables within structured B machines. The case-study serves as a proof of concept that demonstrates the practicality and functionality of our information flow analysis framework. We present the detailed case-study in Chapter 5.

In the following subsection, we present the roadmap for this research work.

1.4.2 Research Plan

To meet the objectives set out in Section 1.3, we plan to realise the following broadly defined outcomes and deliverables, and it is on these deliverables that the table of contents of this thesis is based. Our main contributions are listed as items v, vi, and vii below:

i. Introduction and motivation;

ii. Relevant confidentiality properties;

iii. Programs, refinements and relevant confidentiality-preserving refinements;

iv. Benefits and Limitations of existing confidentiality-preserving refinements;

v. Information flow analysis of standalone B machines, refinements and implementations;

vi. Information flow analysis of structured B machines, refinements and implementations;

vii. A case study: Flow respecting developments in B;

viii. Conclusions and further work.

We now follow with an overview of the layout of the thesis in Section 1.5.

1.5 Structure of Thesis

We structure this thesis roughly on the objectives set out in Section 1.3, following the research plan in Subsection 1.4.2.

Following this introduction, we present in Chapter 2 a detailed literature review of existing work on which our work is based. The chapter is divided into three main sections. The first section (Section 2.2) deals with a review of relevant possibilistic confidentiality properties in the literature. The second section, (Section 2.3), deals with the notion of programs, specifications, and refinements in the literature that we adopt in our work, while the third section, (Section 2.4), presents a brief introduction to the B Method.

In Chapter 3, we present one of our main contributions, which is a framework for analysing intra-machine flows, i.e., the flow of information between variables within standalone B Machines. This chapter presents the core theoretic basis for much of the subsequent work in the thesis, for example, the inter-machine information flow analysis in Chapter 4, and Flow Respecting Developments in B presented in Chapter 5. We also present in Section 3.7 a hypothetical example that illustrates a key advantage of our flow-sensitive information flow analysis over the flow-insensitive ones in the literature, namely: *the ability to correctly analyse a secure program secure even when it contains insecure sub-constructs.* We conclude the thesis with a summary of our accomplishments so far, a discussion of further work and recommendations.

In the meantime, we turn our attention in Chapter 2 to a detailed review of the background work and literature that informs our research work.

Chapter 2

Background and Literature Review

2.1 Introduction

In this chapter we present a number of confidentiality properties in the literature that are relevant to this thesis; the definitions of programs, specifications and refinements that we adopt, and we discuss the difficulties associated with ensuring that confidentiality properties satisfied at earlier stages of the software development process are not lost at later *refined* stages. We focus first on existing work in the field of security properties, discussing the strengths and limitations of the different security properties presented (Section 2.2). We follow with a discussion of definitions of programs, specifications and refinements in the literature (Section 2.3). We conclude this chapter with an introduction to the B Method, showing that even the use of such a strict *formal* specification language as the B Method does not in itself guarantee that security properties satisfied in earlier stages of the development process will be preserved in later (*refined*) stages (Section 2.4).

2.1.1 Confidentiality

The aspect of information flow security we concentrate on in this thesis is *Confidentiality* (aka *Secrecy*), hence subsequent unqualified references to 'security' here is synonymous with 'confidentiality'. The US National Information Systems Security (INFOSEC) Glossary [112] defines confidentiality as: "*Assurance that information is not disclosed to unauthorized persons,*

processes, or devices". Simply put, confidentiality properties define the requirement that secret information is not *read* or *learned* by unauthorised persons, processes, objects or devices either directly or indirectly from permitted observable patterns in multiple executions (or traces) of a program. Generally, in a *Multilevel Secure System* (MLS) [1], a system preserves the confidentiality of a secure entity if the initial value of the secure entity (i.e., the secret object) cannot be learned or deduced from the final values of entities classified lower in the MLS ordering.

The notion of confidentiality can be formally defined using *trace semantics* whereby a trace t is a finite sequence of events in an event system. Given that E denotes a set of events, $l_1, l_2 \in E$ are both low-security events, and $h \in E$ is a secret event. We consider two traces to be *equivalent* from the low security observer's viewpoint, denoted $\approx_L$, if and only if the observer cannot deduce whether or not a high security event occurred in the system. Writing $\textbf{interleave}(\{l_1, h, l_2\})$ to denote all possible executions of program S involving the events l_1, h, l_2 in any order (i.e., $\textbf{interleave}(\{l_1, h, l_2\})$ is a subset of all traces of S, denoted $traces(S)$), the notion that all executions preserve the *confidentiality* of h can be formalised as follows:

$$\begin{aligned}&\forall t \in traces(S),\ h \text{ is secure iff}\\ &\quad \forall t_1 \in \textbf{interleave}(\{l_1, h, l_2\}),\ \exists t_2 \in \textbf{interleave}(\{l_1, l_2\}) \cdot t_1 \approx_L t_2\end{aligned}$$

This notion is graphically illustrated in Figure 2.1, which depicts two different sets of execution sequences of the program S, the first, $traces(S_1)$ includes a high input event h, whereas the second, $traces(S_2)$ only involve low security input events l_1, l_2. For the confidentiality of the high input event, h, to be preserved, the low output events l_1', l_2' in both sets of traces must be low-equivalent, i.e., unaffected by the presence or absence of h.

Having introduced the notion of confidentiality as the aspect of information flow security discussed in this work, we now drill deeper into this concept by

[1] A Multilevel Secure System is a system where the event space (in an event system) or the variables defining the state space (in an input-output system) is partitioned into various security levels, such as 'unclassified', 'confidential', 'secret', 'top secret', for example in the Bell-LaPadula policy used for data confidentiality, or the Biba Integrity policy used for the protection of data integrity.

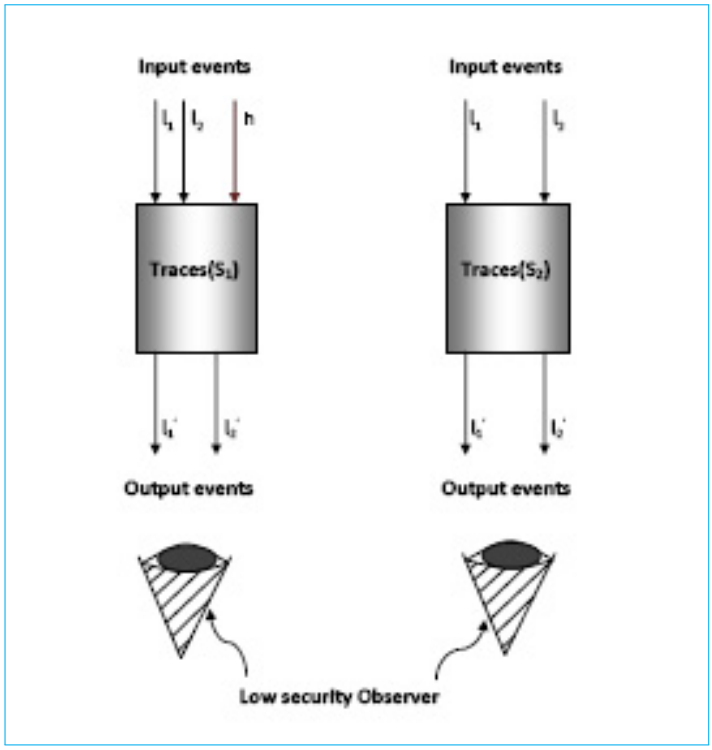


Figure 2.1: Output Events Indistinguishable from Adversary's Observations

discussing the confidentiality properties in the literature that are relevant to our work. This is the subject of Section 2.2.2.

2.2 Confidentiality Properties

By the term *Confidentiality properties*, we refer to the *attribute* of software systems that secret (or *confidential*) information does not end up in the public domain. Hence, in the context of this work, confidentiality really is an *Information Flow Security* property, and generally, such flows could be either *direct* or *indirect* [36]. Transitive flows resulting, for example, from sequential composition are referred to as *indirect* flows. To illustrate, given that x, y, z are variables, ':=' denotes that the variable on the left hand side is updated by the value of the expression on the right hand side, and ';' denotes sequential composition, within the program '$x := y + 1\ ;\ z := x$', y flows into x and x flows into z. Consequently, there is an indirect flow of infor-

mation from y into z. Direct flows are classified as either *explicit* or *implicit* flows [36], [50], [111]. Explicit flows of information are flows resulting from assignments (or simple substitutions, in B parlance), function updates, I/O statements and/or value-returning procedure calls [50]. For example, given that $x := y + z$, the flows from y and z into x are explicit flows. Conversely, a flow is implicit when the value of a variable can be deduced as a result of its use in control flow conditions or guards, for example, when a secret variable is used as a loop guard (or condition) thereby having the updated variable(s) and termination of the program dependent on such a variable [36].

Implicit flows are further subdivided into *local* and *global* flows. When information about the initial value of a variable can be learned or inferred due to dependencies arising when the variable is used in the guard of a conditional (e.g. IF statement) or a guarded substitution, the resulting flow is referred to as a local flow. Consider the program:

$$x := 0 \ ; \ (\textbf{if } y = 4 \textbf{ then } x := 1 \textbf{ else } skip \textbf{ end})$$

In this example, whichever branch of the IF statement is executed, some information about the initial value of y can be learned. An observer can learn that $y \neq 4$ if the final value of x is 0, and he can learn that $y = 4$ if the final value of x is 1. Hence, in either case, an *implicit flow* of information occurs within the *local* IF statement.

Implicit *global* flows occur as a result of information flowing from variables used in the guards of *while* loops or protected substitutions to *all subsequent substitutions* in the program. Given, for example, the program:

$$\textbf{while } (y < 5) \textbf{ do } x := z - 1 \textbf{ end} \ ; \ z := 7$$

Clearly, whenever this program terminates (i.e., the value of z is 7), the observer can conclude that the initial value of y is *greater than or equal to 5*. On the other hand, if the program fails to terminate, the observer can conclude that the initial value of y is in fact *less than 5*. Hence the use of a secret variable in a *while* loop guard *implicitly* leaks some information about

the initial value of the secret variable via the termination behaviour of the loop to all subsequent program points after the *while* loop. So in our example, z depends *implicitly* on y due to *global* flow from the condition $y < 5$ to z in the subsequent substitution $z := 7$. This in an implicit *global* flow because the effect of the flow is not limited to the immediate (or local) *while* loop wherein the guard occurs; rather, it affects all subsequent subprograms within the program.

In the context of this work, we deal with both direct and indirect flows. However, we exclude some types of covert flows[2] such as flows resulting from the monitoring of timing behaviour (i.e., *timing flows*) and resource usage. We deal with the area of security described in the literature as *possibilistic* security properties. Possibilistic security properties encompass security properties that characterise information flow constraints on nondeterministic systems, and do not employ the mathematical concept of probabilities, but rather deal with secrecy in the context of 'sets of possible observations' arising from 'sets of possible executions' of a program. The intuition behind possibilistic security properties is that *if the sets of possible initial and final states of a transition are sufficiently large, it becomes difficult for an observer to deduce with certainty the initial state(s) responsible for any particular transition.* In the following subsection, we address the notion that a specification captures a confidentiality property.

2.2.1 Expressing that a Specification captures a Confidentiality Property

Basically, confidentiality entails the control of *who sees what* during computation. The 'who' in this context could be a system user, a process or other communicating objects within a computer system. The objective is to prevent unauthorised persons, processes, or objects from accessing or deducing information classified as *secret*. We say that a specification captures a confidentiality property when, for all possible animations (or runs) of a specification, there exists no possible trace of the system that allows unauthorised persons, processes, or objects to *learn* the initial value of any specified secret information. In effect, we say such a specification has the

[2] unintended information flow arising from the 'side-effects' of program execution.

'*quality*' or '*attribute*' of security as presented here. Using the term 'entity' in this work to denote *something that exists in its own right in a software system and not merely as part of a bigger thing*, e.g., a variable, file, or other system resources, we can semi-formally state that:

> *A specification satisfies a security property when the specification in question represents mathematically, for all possible runs of the system, the absence of information flow from a secure entity to a public entity through some valid, albeit maliciously contrived, run of the specification.*

It has been shown in the literature that capturing a security property is no easy task [111]. It is even more difficult to guarantee that a security property defined at some stage of the software development process is preserved through refinement(s) to later stages, as we shall be showing later in this chapter. In the following section, however, we discuss confidentiality properties in the literature that are relevant to this thesis.

2.2.2 Relevant Confidentiality Properties

Design details are often left out in the early, more abstract stages of the software development process since the primary aim is to capture the essence of *what* the system is required to do and not *how* the task is to be done. Thus an initial state of the abstract system may correspond to a number of possible final states, such that the final state of any particular transition cannot be determined from any given initial state. This concept is, referred to as *Nondeterminism* (or in B Method parlance, *Underspecification*). The aim of the software development process, in general, is to gradually *refine* away the nondeterminism within a specification until a deterministic implementation is reached, i.e., a specification that cannot be further refined. Consequently, the confidentiality properties we consider relevant to our work are those that relate to Nondeterministic systems, e.g. Generalized Noninterference in the abstract case, and that translate to a corresponding deterministic property such as Noninterference with respect to a concrete implementation of the system.

2.2.2.1 Noninterference (NI)

Noninterference was introduced as a definition of security, i.e., a security property, in the early 1980s by Goguen and Meseguer [61], [62]. The authors formalised what it means for a set of users (or processes, or variables) to be "*noninterfering with*" another set of users (or processes, or variables), i.e., the notion of information not flowing from a high security user to a public user, using a simple general automaton[3]. Even until today noninterference is still the main way of defining security, although further interpretations and extensions of noninterference have been made over the years, for example to make it work with nondeterministic and possibly nonterminating systems, or to make the property composable (or hook-up) [91].

We define our interpretation of NI in this thesis in terms of sets of variables, rather than users. A system satisfies the Noninterference property if a change in the input value of a high security variable does not affect any *future* low-security state of the system. Thus, a low security entity's knowledge of the system remains unchanged notwithstanding any action or inaction of a high security entity within the system [131]. Suppose that l is a low security variable with initial value l_1, and h is a high security variable with values h_1, h_2. For a system in which these variables are defined to satisfy Noninterference, a change in the *initial* value of h from h_1 to h_2 must not affect the final value l_1' of l. This system is noninterfering in the sense that the final value of a low security variable is not *modified or interfered with* by the initial value of a high security variable. This general notion of noninterference is illustrated in Figure 2.2, where $\not\rightarrow$ denotes 'does not flow into', or 'does not interfere with'.

Given that **Ide** denotes the set of variables in a program, S, with the variable space partitioned into a set of low security variables, **L**, and a set of high security variables, **H**. And suppose **Val** denotes a set of variable values, we define a system state, σ, as a function $\sigma : \mathbf{Ide} \rightarrow \mathbf{Val}$. We write $\mathbf{\Sigma}$ to denote a set of states, i.e., $\mathbf{\Sigma} \subseteq \mathcal{P}(\mathbf{Ide} \times \mathbf{Val})$. Let $\mathcal{S}$ be the set of substitutions (or commands) that can modify the system state on variable input. **Out** denotes the visible variable outputs on termination of the system, whereas

[3]An automaton is an abstract mathematical model for a (finite) state machine used for solving computational problems.

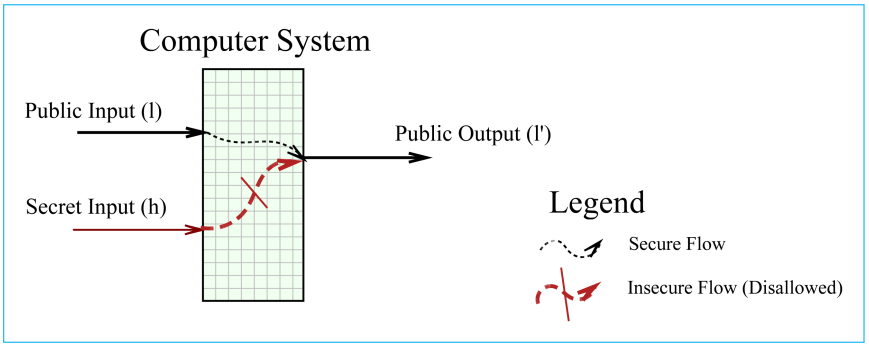


Figure 2.2: Noninterference Illustration

$\mathcal{L}$ stands for the set of security classes to which the variables in **Ide** are mapped. In automata theory, an automaton (or system) requires a *transition relation* which moves the automaton from one state to another when the input string (or sequence) satisfies some *acceptance condition*. Hence, modifying the definition in [62] to relate to variables rather than users, we define a transition relation, *trans*, and an output relation, *out* as follows:

$$\begin{aligned} \mathit{trans} &\in \mathbf{\Sigma} \times \mathbf{Ide} \times \mathcal{S} \to \mathbf{\Sigma} \\ \mathit{out} &\in \mathbf{\Sigma} \times \mathbf{Ide} \to \mathbf{Out} \end{aligned}$$

Notice that *trans* accepts a state, a variable of interest, and a substitution as inputs and moves the automaton to a new state. The intuition behind this is that only one variable may be modified in a system state at each *step-wise* transition. The functional *out*, on the other hand, constructs an output **Out** in a state with respect to a given variable of interest.

Given that $z \in \mathbf{Ide}$ and $s \in \mathcal{S}$, an input string sequence or *word*, denoted $w \in \langle (z,s) \mid (z,s) \in (\mathbf{Ide} \times \mathcal{S}) \rangle$, wherein $(z,s) \in w$ is written, for example, as:

$$w = \langle (z_0, s_0), (z_1, s_1), \ldots, (z_n, s_n) \rangle$$

We write $\mathcal{W} = (\mathbf{Ide} \times \mathcal{S})^*$ to denote the set of all possible sequences of pairs in the input string or *word*, w, i.e., $\{w \mid w \in (\mathbf{Ide} \times \mathcal{S})^*\}$.

We define a *run* or *animation* of the automaton on an input word, w =

$\langle (z_0, s_0), (z_1, s_1), \ldots, (z_n, s_n) \rangle \in \mathcal{W}$, as a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$ where σ_0 is the *start state*, and for all i such that $1 \leq i \leq n$, we have: $\sigma_i = trans(\sigma_{i-1}, (z_i, s_i))$ for all subsequent states. This requires an extension of $trans$ to consume *words* or sequences. This extension we write as $trans^* \in \Sigma \times \mathcal{W}$, which yields the state resulting from applying the inputs one after another in a sequence. (Note: we assume intermediate states are not visible, and not saved internally.) Formally, given that $\hat{}$ denotes sequence concatenation, and assuming that $ws \in \mathcal{W}$ and $w = ws\hat{}(z, s)$, we define $trans^*(\sigma_0, w)$ recursively by cases as follows:

$$trans^*(\sigma_0, w) = \begin{cases} \sigma_0, & if\ w = \langle\rangle \\ trans(trans^*(\sigma_0, ws), (z, s)), & if\ w = ws\hat{}(z, s) \end{cases}$$

We write $[\![w]\!]$ to denote the system behaviour in terms of the sequence of runs of the system on input string w, i.e., $[\![w]\!] = trans^*(\sigma_0, w)$, and $[\![w]\!]_z$ denotes the output on z after executing automaton on w, i.e., $[\![w]\!]_z = out([\![w]\!], z)$. Writing w_H to denote the *word* derived from w by eliminating all pairs (z, s) such that $z \notin \mathbf{H}$, i.e., $w_H = w - \langle (z, s) \mid z \notin \mathbf{H} \rangle$, we formally define w_H structurally by cases as follows:

Given that $ws \in \mathcal{W}$ and $w = ws\hat{}(z, s)$:

$$w_H = \begin{cases} \langle\rangle, & if\ w = \langle\rangle \\ ws_H, & if\ w = ws\hat{}(z, s) \wedge \langle (z, s) \mid z \notin \mathbf{H} \rangle \\ ws_H\ \hat{}(z, s), & if\ w = ws\hat{}(z, s) \wedge \langle (z, s) \mid z \in \mathbf{H} \rangle \end{cases}$$

We now present our interpretation of the noninterference property introduced in [61] in definition 1 below:

Definition 1 (NI)**.** *Given that* $\mathbf{Ide} = \mathbf{H} \cup \mathbf{L}$ *and* $\mathbf{H} \cap \mathbf{L} = \varnothing$*. For all* $z \in \mathbf{L}$ *in all animations of all substitutions of interest, we say that* $\mathbf{H}$ *does not interfere with* z*, written* $\mathbf{H} \not\rightarrow z$*, iff for all* $w \in \mathcal{W}$ *:*

$$[\![w]\!]_z = [\![w_H]\!]_z$$

This interpretation of noninterference basically corresponds to the notion that the output sequence on a set of runs of a system remains unchanged, from the standpoint of a low security variable, on *deletion* of high security variable inputs. Note that the same can also be said of insertion of high

security variable inputs into a *word* or sequence.

Example:
We illustrate our interpretation of Goguen and Meseguer's Noninterference with the following example program, denoted S_1.

IF $p = q$ **THEN**
    $x := y + v$
ELSE
    $x := 0$
END

We assume the variable we are interested in monitoring for information flow in the program is x. Given three input values to x respectively as $x_0 = 5,\ x_1 = 7,\ x_2 = 0$, the pointwise input word to S_1 with respect to x, denoted w, becomes $\langle (x_0, S_1), (x_1, S_1), (x_2, S_1) \rangle$, i.e., $\langle (5, S_1), (7, S_1), (0, S_1) \rangle$.

Now, suppose Table 2.1 describes the states of S_1 corresponding to w.

Initial State	Variable Values
σ_0	$x_0 = 5,\ y = 1,\ v = 7,\ p = 1,\ q = 1$
σ_1	$x_1 = 7,\ y = 2,\ v = 5,\ p = 1,\ q = 0$
σ_2	$x_2 = 0,\ y = 3,\ v = 1,\ p = 0,\ q = 0$

Table 2.1: Initial states of S_1

Given that σ' ranges over final states after each transition, the pointwise transition relation, *trans*, on w using the values given in Table 2.1 with respect to x is given below:

$trans(\sigma_0, x, S_1) = \sigma'_0$,
$trans(\sigma_1, x, S_1) = \sigma'_1$, and
$trans(\sigma_2, x, S_1) = \sigma'_2$

We can then derive the the values of the final states after each transition as

depicted in Table 2.2.

Initial State	Variable Values
σ_0'	$x_0 = 8,\ y = 1,\ v = 7,\ p = 1,\ q = 1$
σ_1'	$x_1 = 0,\ y = 2,\ v = 5,\ p = 1,\ q = 0$
σ_2'	$x_2 = 4,\ y = 3,\ v = 1,\ p = 0,\ q = 0$

Table 2.2: Final states of S_1

Thus the output sequence with respect to x, denoted $[\![w]\!]_x$ is $\langle 8, 0, 4\rangle$.

Given that some other program S_2 with input word $w_2 = \langle (z, S_2)\rangle$ is composed with S_1, such that the input word to the composite program is $w^\frown w_2$, i.e., $w^\frown(z, S_2)$. For the composite system to be noninterfering with respect to x, given that $z \in \mathbf{H}$ is a high security variable, we require that:

$[\![w]\!]_x = \langle 8, 0, 4\rangle$ and $[\![w^\frown(z, S_2)]\!]_x = \langle 8, 0, 4\rangle$.

This illustrates that the insertion of a high security variable input value after a sequence of inputs does not interfere with any observable low security variable output of the system.

For a multilevel secure (MLS) system, noninterference can be interpreted as follows:

Given that $\mathcal{L} = \langle \mathcal{L}, \leq\rangle$ is a multilevel security lattice;

A function, $classify \in \mathbf{Ide} \to \mathcal{L}$ maps variables to their respective security classes;

And for any $l \in \mathcal{L}$, we define a downward inclusion ordering on the variable space, which collects the set of variables with security classification less than or equal to l. This intuitively collects the variables that may be read by variables with security classification l or higher:

Given that $x \in \mathbf{Ide},\ k \in \mathcal{L},$ and $classify(x) = k$, the downward inclusion ordering, $\downarrow x$, is defined as:

$\downarrow x = \{z \in \mathbf{Ide} \mid classify(z) \leq k\}$.

Similarly, an upward exclusion ordering is defined, which collects the set of variables with security classes higher than the security level of interest. Formally, the upward exclusion ordering with respect to $x \in \mathbf{Ide}$, denoted $\uparrow x$ is defined as:

$$\uparrow x = \{z \in \mathbf{Ide} \mid classify(z) \not\leq k\}$$

Hence, we say a system is MLS secure iff $\forall l \in \mathcal{L}$:
if $x, y \in \mathbf{Ide}$ and $classify(x) = l$,
then $y \in \uparrow x$ *implies* $y \not\rightarrow x$

The interpretation of NI described in this section is illustrated in Figure 2.3, which shows that a change in the input value of high security variables does not impact the output on low security variables. In the figure, we depict three runs S_{r1}, S_{r2}, S_{r3} of the system S. The first run has no input from a high security variable; the second has a secret input h_1; and the third run has the secret input changed to h_2. $\mathbf{L}$ denotes a set of low security variables, and a fixed input is passed to each of the variables in $\mathbf{L}$ in all three runs of S. In all three runs of S, the output visible to a low security observer on l' must all be equivalent. In other words, the output visible to low security observers are *consistent* in all runs of the system, whatever the input string passed to the system. This corresponds to what Zakinthinos and Lee termed a *Low Level Equivalent Set* LLES [84].

McLean in [98] points out that Noninterference deals with some of the limitations of access control (e.g., by handling covert flows and read and write dependencies better than access control mechanisms) and that Noninterference states a security requirement rather than a method for meeting that requirement. While this is useful in itself in that it expands our understanding of security properties, developing methods for enabling programs meet this requirement is even more beneficial. This is one of the main motivations behind our work on the development of a framework for analysing B Machines to ensure that the operations defined in the machines meet the

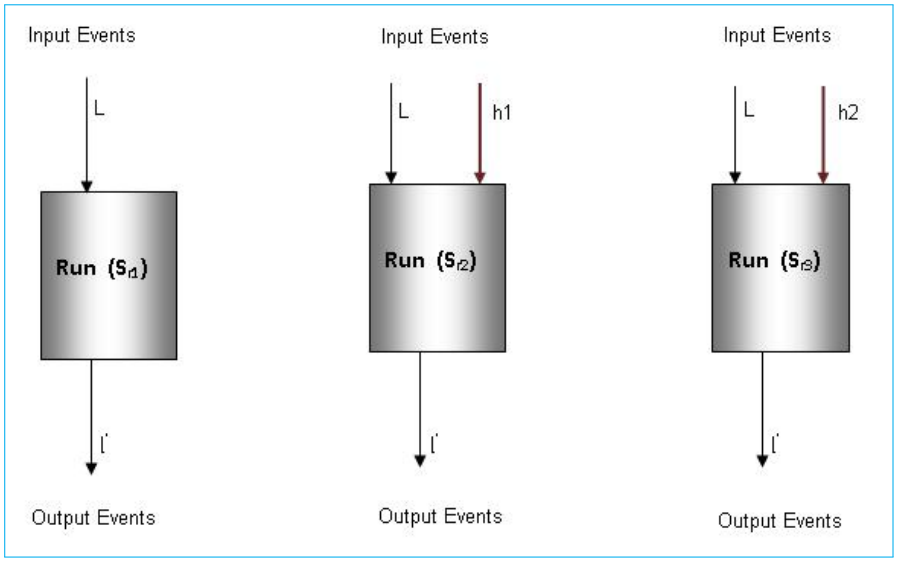


Figure 2.3: Illustration of Noninterference 'Semantics'

Noninterference property[4].

Beneficial though Noninterference has proved to be, it is not without its limitations, some of which are discussed below.

- Zakinthinos and Lee in [84] noted that '*Goguen and Meseguer's original definition of Noninterference was only applicable to deterministic systems*'. Sabelfield and Myers in [111], and McCullough in [91] are some other notable experts to corroborate this claim. Hence this inability to handle nondeterministic information flow where the output is not necessarily a function of the input sequence, i.e., different outputs may result from the same input sequence is a limitation of noninterference. For example, the tossing of a coin may result in a 'head' in one instance, and a 'tail' in another, hence the output is not a function of the input.

- Inability to handle interrupts and nontermination. The former follows because noninterference is defined in terms of inputs and outputs, without recourse to concurrency or multitasking [91]. The simple fact

[4]We hope in the future to be able to extend the framework to analyse machines satisfying other possibilistic security properties in the literature.

that if a program *aborts*, no output is guaranteed means that noninterference does not account for nontermination.

- Composability is only possible under extremely impractical assumptions such as that all buffers are unbounded, and that there must be no merging of outputs (as in parallel processing of inputs, thereby losing the benefits of parallel processing) [90], [91].

- Another documented limitation of noninterference in the literature is that it is too restrictive, and it does not take account of *information release* whereas many real world systems are intended to leak some kind of information. Flow sensitive information flow analysis frameworks such as the ones introduced by Hunt and Sands [75], Amtoft and Banerjee [8], and Clark et al [36], all of which are precursors to the approach in this thesis mitigates against this limitation of noninterference. We do not explore this limitation in detail because the area concerns *declassification*, which is beyond the scope of this thesis.

Consequently, a more generalised form of NI, termed Generalized Noninterference, that deals with nondeterminism, and corresponds to NI in the deterministic case, was introduced by McCullough in [89] as an extension to Sutherland's earlier work on Nondeducibility [131], and this is the subject of the following section.

2.2.2.2 Generalized Noninterference (GNI) and Weak Noninterference (WNI)

If for every valid trace (sequence of inputs and outputs) of a system and every possible alteration to that trace (such as by deleting or inserting high security inputs) the set of possible low security futures after the modification is equal to the set of possible low security futures before the modification, then the system is said to satisfy the Generalized Noninterference property [89][90]. Hence, a low security observer cannot, by observing the set of possible low security futures, rule out any possible input sequence involving high security variables. GNI extends NI into the realms of Nondeterministic and Interruptible systems, where the outputs may depend as much on the interleaving of the inputs and outputs as on the sequence of inputs.

To formalise GNI, we lift the definition of NI given in definition 1 (on the effect of some input sequence on the observable future values of low security variables) to the effect of sets of sequences of inputs on sets of observable future values of low security variables. To that end, we assume a set of input sequences, $\mathbf{W} \subseteq (\mathbf{Ide} \times \mathcal{S})^*$, defined as $\mathbf{W} = \{w_1, w_2, \ldots, w_n\}$. Similarly, $\mathbf{W}, \langle (x, s_h) \rangle$ means $\mathbf{W}$ followed by the singleton input sequence $\langle (x, s_h) \rangle$. We write $[\![\mathbf{W}]\!]$ to denote the set of all runs of the system on $\mathbf{W}$ with each run starting at σ_0, i.e., $\forall w \in \mathbf{W},\ trans^*(\sigma_0, w)$. This could be expanded as $[\![\mathbf{W}]\!] = \{trans^*(\sigma_0, w_1),\ trans^*(\sigma_0, w_2),\ \ldots,\ trans^*(\sigma_0, w_n)\})$. Correspondingly, we write $[\![\mathbf{W}]\!]_z$ to denote the set of observable future values of z for all runs of the system on $\mathbf{W}$, i.e., $\forall w \in \mathbf{W},\ [\![\mathbf{W}]\!]_z = out([\![\mathbf{w}]\!], z)$. Finally, we write $\mathbf{W_H}$ as an extension of w_H defined earlier in section 2.2.2.1 as: $\mathbf{W_H} \triangleq \forall w \in \mathbf{W} \cdot w_H$.

Definition 2 (GNI)**.** *Given a system S with $\mathbf{Ide} \supseteq \mathbf{H} \cup \mathbf{L}$ and $\mathbf{H} \cap \mathbf{L} = \varnothing$. Suppose $s_h \in \mathcal{S}$ is a substitution that updates a high security variable. For all $x \in \mathbf{H}$ and all $z \in \mathbf{L}$, we say that S satisfies GNI with respect to x, i.e., $\mathbf{H}$ does not GNI-interfere with z, written $\mathbf{H} \not\rightarrow_{GNI} z$, iff for all $\mathbf{W} \subseteq \mathcal{W}$:*

$$\forall [\![\mathbf{W}, \langle (x, s_h) \rangle]\!]\ ,\ \exists [\![\mathbf{W'}]\!] \cdot [\![\mathbf{W'}]\!]_z = [\![\mathbf{W}, \langle (x, s_h) \rangle]\!]_z$$

This definition simply states that *given any possible set of runs on any set of valid input strings (high security variables inclusive), i.e. $[\![\mathbf{W}, \langle (x, s_h) \rangle]\!]$, it is always possible to construct another set of runs on sets of low security input variable strings only, $[\![\mathbf{W}]\!]$, that yields a consistent set of observable output values with respect to the set of low security input strings* $\mathbf{W}$. Hence, by observing $[\![\mathbf{W}]\!]$ and (for all $z \in \mathbf{L}$) $[\![\mathbf{W'}]\!]_z$, the low security observer is unable to determine whether there was an initial high security variable input to S or not, i.e., he cannot tell whether the output he is seeing is produced by $[\![\mathbf{W}, \langle (x, s_h) \rangle]\!]$ or $[\![\mathbf{W}]\!]$.

Figure 2.4 illustrates the notion of GNI presented in Definition 2. This shows that a set of sequences of low security inputs, $\mathbf{W}$, followed by a sequence of high security input(s), $\langle (x, s_h) \rangle$, (shown in dotted lines to indicate input sequence is *hidden* or *secret*) yields an output, $[\![\mathbf{W}, \langle (x, s_h) \rangle]\!]_z$, consistent with the output of a set of sequences of inputs with the hidden input se-

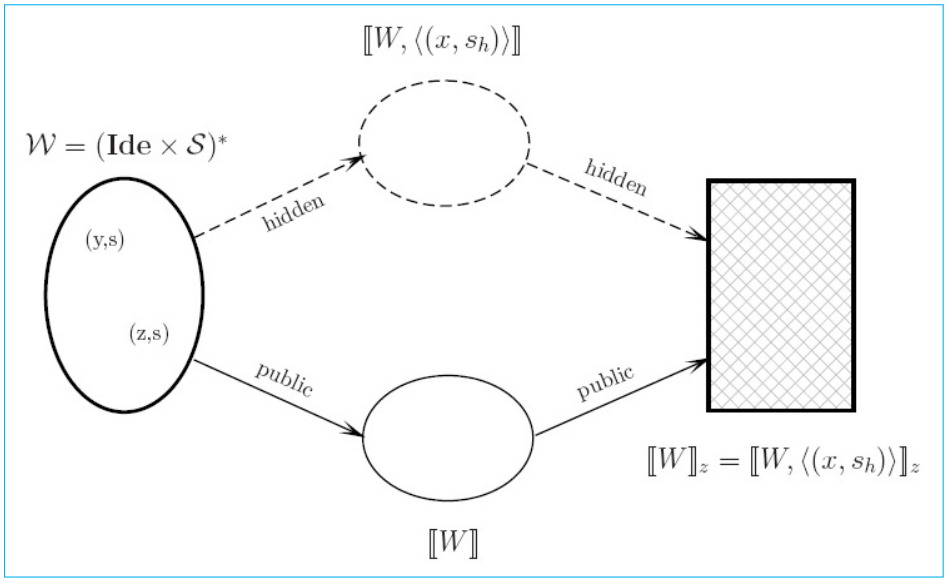


Figure 2.4: Illustration of GNI 'Semantics'

quence removed, $[\![\mathbf{W}]\!]_z$.

Although GNI makes Noninterference more useful by dealing with sets of possibilities and nontermination, GNI also has its limitations. For example, it is generally not composable especially for systems with feedback, i.e., two-way communication between processes and it could suffer the infamous refinement paradox, which will be discussed in more detail in Section 2.3.5.2.

A somewhat similar extension of NI into the realms of Nondeterminism is *Weak Noninterference* WNI, which allows the removal of all high security inputs without impacting the low security behaviour of the system [87]. Recall from the notation defined earlier that $\forall z \in \mathbf{Ide}$ and $s \in \mathcal{S}$, an input string sequence is given as $w = \langle (z_0, s_0), (z_1, s_1), \ldots, (z_n, s_n) \rangle$. We extend this notation by defining $dom(w)$, for use in our interpretation of WNI, as:

$$dom(w) = \{x \in \mathbf{Ide} \mid (\exists s \in \mathcal{S})\ (x, s) \in w\}$$

Definition 3 (WNI)**.** *We say a system S satisfies Weak Noninterference with respect to some input on x if and only if for every input sequence to*

S, from the view of some variable y with security classification lower than x's, there is another input sequence on y, which exhibits the same visible behaviour (output) as the input on x.

$$\forall w \in \mathcal{W},\ \exists w' \in \mathcal{W}\ \cdot\ \forall x \in dom(w),$$
$$\exists y \in dom(w')\ \cdot\ y \notin \uparrow x \wedge [\![w]\!]_y = [\![w']\!]_y$$

This is a formal way of saying that the set of possible behaviours of a system with respect to some variable remains unchanged even when the inputs on high security variables in the system are removed.

One advantage of WNI is that responses to inputs are not assumed to be instantaneous (i.e., delays can be handled securely) [87]. Following from that, termination is not assumed. This is the main difference between WNI and Strong Noninterference (SNI) because SNI is termination-sensitive whereas WNI is not. Trivially, though, SNI implies WNI, i.e., any system that satisfies SNI also satisfies WNI, but not vice versa. For example, two program statements are equivalent (in both SNI and WNI) if they both halt or both diverge at equivalent values. If one statement halts (or diverges) while the other does not, however, the two statements are equivalent under Weak Noninterference, but not under Strong Noninterference. Since Strong Noninterference is less useful than GNI and WNI in that it prevents systems that are obviously secure such as update flows (i.e., low writing to high), and it permits intuitively insecure systems [62], we will be employing the intuitions behind GNI and WNI rather than SNI in our work.

McCullough in [89], [90] further extended his earlier work on GNI in order to overcome some of its limitations. This resulted in the security property: 'Restrictiveness', which is the subject of the following section.

2.2.2.3 Restrictiveness (RES)

Restrictiveness is a hook-up (or composable) security property that is viewed by some as a stronger version of Noninterference, sometimes classified as Strong Noninterference [91]. This is a further development of McCullough's work on GNI [89] and Sutherland's work on Nondeducibility [131] neither of which are composable. McCullough argued that although a system sat-

isfying either GNI or Nondeducibility guarantees that a variation in high security input does not interfere with future low security behaviour, a pair (or composition) of a high security input followed immediately by a low security input is not guaranteed to have the same effect. Hence RES was introduced to *make* security properties "*Hook-up*", i.e., composable by restricting messages so that insertion of a new high security output before or during the sequence of inputs after the *first* change in high security input is rejected [38]. McCullough asserted that it is sufficient for a state machine to satisfy the following conditions for it to be restrictive with respect to a set of low security events.

- The machine is input-total, i.e., for all *reachable* states in a system, every possible input must be able to transition the system from *at least* one state to another [56], i.e., there is to be no blocking on inputs. Given that $\mathbf{\Sigma}$ denotes the set of states in the system, $\mathbf{E_I}$ denotes the set of input events, and $\stackrel{e}{\rightarrow}$ denotes state transition by input event e, the notion of input totality is defined using state transition system notation as shown in Formula 2.1 below [87].

$$\forall \sigma \in \mathbf{\Sigma} \ \cdot \ \forall e \in \mathbf{E_I} \ \cdot \ \exists \sigma' \cdot \sigma \stackrel{e}{\rightarrow} \sigma' \wedge \sigma' \in \mathbf{\Sigma} \tag{2.1}$$

  Formula 2.1 is based on the notion that every state set, $\mathbf{\Sigma}$, of a system may always be extended with an input event, e, if the system is input-total [84]. Hence a system that is input-total is also closed with respect to all inputs satisfying the property.

- When two processes correspond to the same low security state, there is an equivalence relation between the states of both processes. Given, for example that $\mathbf{HI}, \mathbf{HO}, \mathbf{LI}, \mathbf{LO}$ denotes set of high-level inputs, high-level outputs, low-level inputs and low-level outputs respectively; S_1, S_2 denote instances of the system under consideration, and $\equiv_L$ denotes 'low-level equivalence'. Using the notation introduced here, the conditions for this low-level equivalence relation are summed up in the following statements, namely:

  - $\forall h_i \in \mathbf{HI} \ \wedge \ S_1 \stackrel{h_i}{\rightarrow} S_1', S_1 \equiv_L S_1'$.

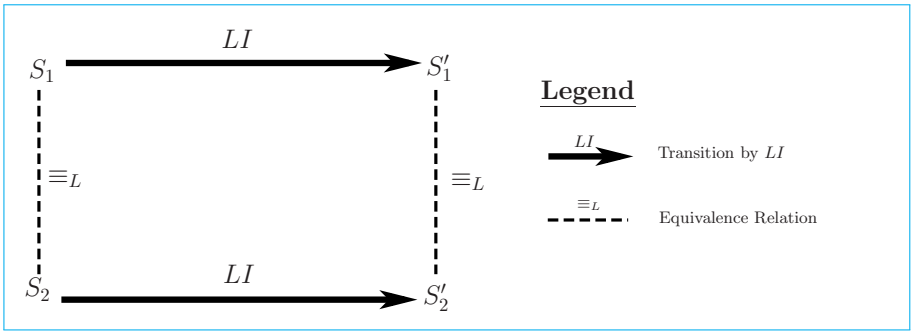


Figure 2.5: Restrictiveness Condition $\forall l_I \in \mathbf{LI}$

- $\forall l_i \in \mathbf{LI} \wedge S_1 \xrightarrow{l_i} S'_1,\ if\ S_1 \equiv_L S_2$ then
  $\exists s \in S'_2 \cdot S'_2 \equiv_L S'_1 \Rightarrow S_2 \xrightarrow{l_i} S'_2$ - - see Figure 2.5
- $\forall ho \in \mathbf{HO} \wedge S_1 \xrightarrow{ho} S'_1,$ if $S_1 \equiv_L S_2$ then
  $\exists s \in S'_2 \cdot S'_2 \equiv_L S'_1 \wedge \exists ho' \cdot S_2 \xrightarrow{ho'} S'_2$ - - see Figure 2.6
- $\forall l_o \in \mathbf{LO}, \forall (ho_1, ho_2) \in \mathbf{HO},$ if $e_n = \{ho_1\} \cup \{l_o\} \cup \{ho_2\} \cdot S_1 \xrightarrow{e_n} S'_1,$
  $\exists (ho'_1, ho'_2, S'_2)\ \cdot S'_2 \equiv_L S'_1 \Rightarrow S_2 \xrightarrow{e_n} S'_2$

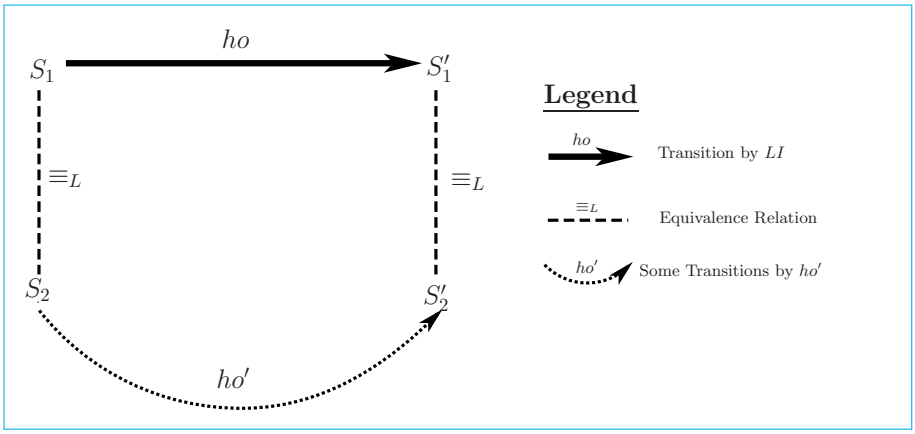


Figure 2.6: Restrictiveness Condition $\forall h_o \in \mathbf{HO}$

Although RES is composable and deals with Nondeterministic systems, interrupts and nontermination, it unfortunately is not preserved when the Nondeterminism is refined away to get a deterministic implementation of the system. This is a problem we will address in Chapters 3 and 4 of this

thesis. In the meantime, we will now review Sutherland's Nondeducibility in the following section.

2.2.2.4 Nondeducibility (ND)

Sutherland in [131] proposed Nondeducibility to capture the essence that a low security user should not be able to deduce or learn to *a high level of certainty* anything about the activities of high security users by observing events legitimately visible to him (i.e., the low security user). Thus, for every trace of the system, there is a second trace, from the low security (l) user's view, which contains only the input and output traces of events with security classification lower than or equal to l. Sutherland's framework is based on a function mapping a set of *possible worlds* or traces to a set of values, $\mathbf{V}$. To enhance comparison with our language-based, program analysis approach to the prevention of insecure flows (Chapter 3), we re-present Sutherland's Nondeducibility as functions on 'sets of states' rather than execution sequences (or traces).

Definition 4 (ND)**.** *Given that* $\mathbf{\Sigma}$ *denotes the set of system states, and let* **Ide** *and* **Val** *be the set of variables and the set of values respectively.* **Ide** *is partitioned into* $\mathbf{H} \subseteq \mathbf{Ide}$ *and* $\mathbf{L} \subseteq \mathbf{Ide}$ *corresponding to the set of high security variables and the set of low security variables respectively. We define the set of* ***possible worlds*** *as* $\mathbf{\Sigma} \subseteq \mathcal{P}(\mathbf{Ide} \rightarrow \mathbf{Val})$*. We say that a system fails Nondeducibility if and only if for all states where valuations of low security variables are unchanged, there exists a valuation of high security variables that affects the low security observations of the system.*

Note: For simplicity, we write $\sigma(x) \cup \sigma(y)$ to denote a set containing the values of the variables x and y in the state σ. We say a system S fails ND if:

$$\begin{aligned} &\forall \sigma \in \mathbf{\Sigma} \cdot \forall x \in \mathbf{L} \text{ Given that: } y_1 \in \mathbf{H} \wedge \mathbf{V} \subseteq \mathbf{Val}, \\ &\qquad if\ \sigma(x) \cup \sigma(y_1) = \mathbf{V} \text{ then } \exists y_2 \in \mathbf{H} \cdot y_1 \neq y_2 \wedge \sigma(x) \cup \sigma(y_2) \neq \mathbf{V} \end{aligned} \tag{2.2}$$

The formal definition in Formula 2.2 above shows that a change in the value of a high security variable is deducible since $\sigma(x) \cup \sigma(y_1) = \mathbf{V} \Rightarrow \exists y_2 \in \mathbf{H} \cdot \sigma(x) \cup \sigma(y_2) \neq \mathbf{V}$. One consequence of this formal definition is that it

shows ND as a 'one-way' property, i.e., it only constrains read-ups[5]. Thus safe flows, e.g. auditing flows (read-downs[6] from **H** to **L**) and update flows (i.e., write-ups[7] from **L** to **H**) are disallowed [38].

As pointed out by McCullough in [90], ND is *weaker* and more general than NI in that a system that satisfies ND would also satisfy NI (but not vice versa). Furthermore, ND is a possibilistic security property and it does not require systems to be uninterruptible and deterministic as NI does. However, the same author pointed out that a system that satisfies GNI also satisfies ND, making GNI an even weaker property than ND. Much like GNI, however, Nondeducibility also is not preserved under composition.

2.2.2.5 Nondeducibility on Strategies (NDoS)

Introduced by Wittbold and Johnson [81] as an extension of Nondeducibility, Nondeducibility on Strategies was formalised to overcome some of the limitations of Nondeducibility, namely:

- Solving the problem of implicit flows due to *cooperating strategies* between two entities to leak information via side-effects of system behaviour rather than leakage through deduction based on input strings. This is a problem not handled by Nondeducibility, where a system that is ND secure on a set of input strings may fail ND on a set of transmitter input strategies.
- Sutherland's Nondeducibilty property is designed to work with systems without feedback. This is another limitation addressed by [81] in NDoS.

NDoS is formalised on a *synchronized state machine*[8] where the transmitter and the receiver may collude on strategies designed to leak secret information without such information being directly deducible from transmitter input strings. For example, a transmitter may intentionally cause some

[5]Read-Ups - ability of an object to read or deduce information at higher security levels
[6]Read-Downs - ability of an object to read or deduce lower security information
[7]Write-ups - ability of an object to write information to higher security objects
[8]Synchronized state machine - defined in [81] as a state machine wherein the inputs, the outputs and the state transitions are synchronised.

system violations, such as *buffer overflows* as a cue to the receiver that a secret action has taken place. So when the receiver observes such violations (which are often caught as *exceptions*), he can deduce that a high event has occurred, not because of having been directly able to deduce such information because of transmitter input string, but rather because he is able to deduce the information via *transmitter strategy*. This way, secret information could be *noiselessly* transmitted to a low security observer. Intuitively, a *strategy* is a scheme (pre-arranged between sender and receiver) that allows secret input to be deducible as a collection of functions of a given history of the system's inputs and outputs. Hence, a strategy maps *history* to the next inputs. This is the weakness often exploited by Trojan Horse attacks.

Given that U denotes a user process; $\mathbf{I_U}, \mathbf{O_U}$ denotes set of inputs and set of outputs respectively to the user process U. Let π denote a strategy. Formally, a strategy could be defined as follows:

Definition 5 (Strategy)**.** *Given that ι range over counting numbers up to n, i.e., $1 \leq \iota \leq n$, a strategy of length n for a user process U is a sequence of n functions, $\pi = (\pi^1, \ldots, \pi^n)$ such that:*

$$\pi^\iota \in (\mathbf{I_U} \times \mathbf{O_U})^{\iota-1} \rightarrow \mathbf{I_U}$$

Given that λ denotes the low security user's view of length n, the authors defined NDoS as shown below:

Definition 6 (Nondeducibility on Strategies)**.** *A synchronized state machine is NDoS iff, for any n, the low security user's view, λ, of length n is consistent with any high transmitter strategy, π, of the same length n.*

This definition is based on the intuition that whatever view of the system the low security user has, he is unable to rule out any strategy of the high security user. Despite the many benefits of NDoS over ND, though, NDoS still has its own limitations. For example, as acknowledged even by its proponents [81], NDoS still suffers from the difficulties of combinatorial theories.

We proceed in the next section with our review of O'Halloran's Noninference, and it's extension by McLean to Generalized Noninference.

2.2.2.6 Noninference (NInf) and Generalized Noninference (GN)

O'Halloran introduced the notion of Noninference in [116] as a means of ensuring high security activity is not *inferable* by low security observers. The intuition behind NInf is that high security activity is not inferable by low observers if both activities are separated, one from the other. Using the trace semantics in which NInf was originally defined, the idea is that the removal of all high security activities from any trace of the system yields a trace that is itself also a valid trace of the system. In other words, the trace of a system satisfying NInf is *closed* under a function that removes all high security activities from the original trace set. This intuition corresponds in our state-based framework to stating that the removal of all valuations of high security variables from the set of all valid system states yield some valid system states that are *low-equivalent* to the original set of states. Recall that we defined the set of all possible input strings of a system S as $\mathcal{W} = (\mathbf{Ide} \times \mathcal{S})^*$. We hold that S is *low-equivalent* on input strings to some set of runs of the system that are consistent with the observable runs of the system with respect to a low security variable. We formalise this notion, denoted $\approx_L$ as follows:

Given that $w \in \mathcal{W}$ and for all $(z, s) \in w$ such that $z \in \mathbf{L}$,
$$w \approx_L w' \;\triangleq\; w, w' \in \mathcal{W} \wedge [\![w]\!] = [\![w']\!]$$

We give a formal definition for NInf in terms of $\approx_L$ below:

Given that $w \in \mathcal{W},\ z \in \mathbf{L}$ and $x \in \mathbf{H}$, we write $w \in \approx_L$ iff
$$\exists w' \in \mathcal{W} \;\cdot\; [\![w']\!]_z \;\wedge\; w'_H = \langle\rangle \wedge [\![w'_H]\!]_x = \langle\rangle$$

The main problem with Noninference is that it is simply *too strong* since it disallows output on high security variables, $[\![w'_H]\!]_x$. Thus, it does not account for information flow from a low security object to a high security object, i.e., it blocks safe flows like write-ups. NInf is somewhat like Separability in its strict separation of high security activities from low security activities, hence it also has many of the limitations of Separability. Like Noninterference, NInf also lacks the capability to deal with Nondeterministic systems. Hence McLean in [97] extends this property, calling the new (*extended*) property Generalized Noninference (GN). Rather than seek to

eliminate all high security activities, GN only requires that when high security *inputs*, **HI**, are removed from any trace, the resultant trace must be a valid trace as well as be equivalent to the low security values of the original trace. This corresponds in our state-based framework to stating that the removal of all valuations of high security inputs from the set of all valid system states yields some valid system states that are *low-equivalent* to the original set of states. One of the improvements of GN over NInf is that flows from low security objects to high security objects are captured, hence GN is weaker than NInf in that the state space captured by GN is larger than that captured by the latter, i.e., NInf *implies* GN but not vice-versa. A formal definition of GN is given below.

Given that $w \in \mathcal{W}$ and $z \in \mathbf{L}$, we write $w \in\!\approx_L$ iff
$$\exists w' \in \mathcal{W} \cdot [\![w']\!]_z \ \wedge \ w'_H = \langle\rangle$$

Notice in our characterisation of GN that only input sequences involving high security variables, w'_H are disallowed. We now turn our attention, in the following section, to the notion of 'Separability'.

2.2.2.7 Separability (SEP)

In the the late 1970s / early 1980s, the mathematical concept of Separability was employed in the design of secure operating systems [126],[127], whereby a process is said to be *Separability* secure if it can be constructed as a parallel composition of separate components with disjoint alphabets. Because such components do not synchronise on events (since alphabets are disjoint), there clearly is no chance of any kind of information flow between them. In *process algebra* talk, using Hoare logic [73], we denote by αP the alphabet (i.e., possible events) of a process P. Assuming αP is partitioned into $\{A, B\}$, and there are sub-processes P_A, P_B with $\alpha P_A = A$ and $\alpha P_B = B$, then, writing $P \parallel_\varnothing Q$ to denote the parallel composition of processes P and Q with no synchronisation of events, we reference Jacob's definition in [79] that P is *separable* with respect to $\{A, B\}$ only if $P = P_A \parallel_\varnothing P_B$. Formally, writing $\Subset$ to denote 'sub-process of', we state the notion of Separability as:

$$\forall P \cdot (P_A, P_B) \Subset P \cdot (\alpha P_A \cap \alpha P_B = \varnothing) \Rightarrow P = P_A \parallel_\varnothing P_B \qquad (2.3)$$

The *Separability* Formula in 2.3 simply states that "*a process is Separable if it is equivalent to a parallel composition of sub-processes with disjoint alphabets*". Hence, every possible sequence of high security events is possible with any observation by a low security adversary [84]. This definition shows that rather than being a security property guaranteeing the absence of *secure* information flow between system states, SEP is more like two or more non-communicating, non-synchronising systems, which do not allow any possibilistic flow of information between high security objects and low security objects. Before we define our representation of SEP, we first define our representation of the notion of *Low Level Equivalent Set* (LLES) introduced by [84]. Recall that we write $w \in \mathcal{W}$ to denote a *word* or sequence. Suppose for any $w \in \mathcal{W}$, w_L denotes w with only the set of low security events retained, i.e. $w_L = w - \langle (z, s) | z \notin \mathbf{L} \rangle$. Given a system S, we write $\mathbf{LLES}(w, S)$ to denote the set of words with the same low security events as w in the same order, i.e.:

$$\mathbf{LLES}(w, S) = \{u \mid (u \in \mathcal{W})\ w_L = u_L\}$$

Using our notion of **LLES** defined above, Zakinthinos and Lee's definition of Separability in [84] could be formalised as shown in Formula 2.4 below. This simply states that no behaviour can be ruled out with the occurrence of any high security event.

$$\begin{aligned} \forall w \in \mathcal{W} \cdot w_L \in \mathbf{LLES}(w, S)\ \wedge\ \forall u, v \in \mathcal{W} \cdot u^\frown v \in \mathbf{LLES}(w, S)\ \wedge \\ v_H = \langle\rangle,\ \forall \alpha \in \mathbf{H} \cdot (t \in \mathcal{W} \cdot u^\frown \langle \alpha \rangle_H = t_H) \Rightarrow \\ u^\frown \langle \alpha \rangle^\frown v \in \mathbf{LLES}(w, S) \end{aligned} \tag{2.4}$$

Notice that $v_H = \langle\rangle$ indicates that the word v has no high security events, whereas $u^\frown \langle \alpha \rangle_H$ definitely has at least one high security event, α. This definition of SEP, as pointed out by [84], requires that all high level outputs be *possible* in all runs of a system.

The fact that Separability rejects even secure flows from low security objects to high security objects is one of its primary limitations. In a sense, SEP is both *too weak* and it gives no guarantee of Noninterference, and *too strong* in that it prevents intuitively safe flows [84]. Hence, in as much as Separability does not allow any interaction or information flow whatsoever

between low security and high security entities, it is the *strongest* and least useful of all Confidentiality properties, as it could be employed only in very limited situations.

In the following section, we present the notion of Perfect Security Property in the literature and show its relationship with Separability.

2.2.2.8 Perfect Security Property (PSP)

Separability has been touted as an example of perfect security in that it allows no interaction between low and high security objects [84]. To overcome some of the limitations of Separability, Zakinthinos and Lee [84] introduced the notion of *Perfect Security Property*, which involves a weakening of the *separation* between low and high security objects such that information could flow *up* from low security objects to high security objects, but not *down* in the opposite direction [99]. The intuition behind PSP is that an observer with knowledge of the system specification cannot, by observing the non-secure information in the system, draw any conclusions about operations in the secure domain of the system, i.e., when a high security event occurs, the low security view of the system is always consistent with some possible low level event output, so that from the latter's viewpoint, it is as if nothing has occurred in the secure domain of the system.

With a slight modification of the definition of Separability given in Formula 2.4, we present below a formal definition of PSP.

$$\begin{aligned} \forall w \in \mathcal{W} \cdot w_L \in \mathbf{LLES}(w, S) \ \wedge \ \forall u, v \in \mathcal{W} \cdot u^\frown v \in \mathbf{LLES}(w, S) \ \wedge \\ v_H = \langle\rangle, \ \forall \alpha \in \mathbf{H} \cdot u^\frown\langle\alpha\rangle \in \mathcal{W} \Rightarrow u^\frown\langle\alpha\rangle^\frown v \in \mathbf{LLES}(w, S) \end{aligned} \tag{2.5}$$

The crucial difference between [84]'s definition of SEP and PSP is that while the former is dependent only on high security events (i.e., definition retricted to **H**), the latter depends on all event sequence.

Mantel in [99] extended the concept of PSP in [84] by employing *State Event Systems* (abbreviated SES) and used this concept to formalise two *unwinding conditions* which, if satisfied by local instances of an event system,

would guarantee that PSP is also satisfied. The set of states from which an event is permitted to execute then becomes the precondition, whereas the postcondition is defined as a function from the *precondition* to the set of possible states resulting from execution of the event. This makes it easier to prove PSP as it is formalised in terms of local events rather than sets of events [99], [84].

Before detailing how SES works, however, [99] introduced a definition of security as a *flow policy* on a set $\mathcal{D}$ of domains of security classes. The various flows the policies are based on are itemised below:

- Noninterference, denoted '$\not\rightarrow$', is defined as $\not\rightarrow \subseteq \mathcal{D} \times \mathcal{D}$ meaning that for all $\mathbf{H}, \mathbf{L} \subseteq \mathcal{D}$, $\mathbf{H} \not\rightarrow \mathbf{L}$ means that *no information flow* is allowed from $\mathbf{H}$ to $\mathbf{L}$.
- '$\rightsquigarrow_V \subseteq \mathcal{D} \times \mathcal{D}$' expresses visible information whereby given that $\mathbf{H}, \mathbf{L} \subseteq \mathcal{D}$, the authors write $\mathbf{L} \rightsquigarrow_V \mathbf{H}$ to mean that information in domain $\mathbf{L}$ is visible to $\mathbf{H}$.
- *Don't care Deducibility.* We use this phrase to describe the author's flow condition that for all $\mathbf{H}, \mathbf{L} \subseteq \mathcal{D}$, information shall not flow from $\mathbf{H}$ to $\mathbf{L}$, although we do not care if information in $\mathbf{H}$ can be deduced from information visible to $\mathbf{L}$. Where $\rightsquigarrow_N \subseteq \mathcal{D} \times \mathcal{D}$, this flow condition is written as $\mathbf{H} \rightsquigarrow_N \mathbf{L}$,

With these flow conditions, [99] then defined a flow policy as a tuple $(\mathcal{D}, \rightsquigarrow_V, \rightsquigarrow_N, \not\rightarrow)$. Hence to define a flow policy that specifies flow conditions $\rightsquigarrow_V$ and $\not\rightarrow$, while assuming the condition $\rightsquigarrow_N$ is empty, [99] used notation of the form $S_{\mathbf{L},\emptyset,\mathbf{H}}$ [9].

Given that $\mathbf{\Sigma}$ denotes a set of states; $\mathbf{\Sigma_I} \subseteq \mathbf{\Sigma}$ denotes set of initial states; $\mathbf{T_R} \subseteq \mathbf{\Sigma} \times \mathbf{E} \times \mathbf{\Sigma}$ denotes the set of transitions of the *event system* from one state to another as a result of the execution of an event or sequence of events, $\mathbf{E}$. $\mathbf{E}$ denotes a set of events; $\mathbf{I} \subseteq \mathbf{E}$, $\mathbf{O} \subseteq \mathbf{E}$ denotes input and output events respectively. A state event system is configured as a quintuple $(\mathbf{\Sigma}, \mathbf{\Sigma_I}, \mathbf{E}, \mathbf{I}, \mathbf{O}, \mathbf{T_R})$. Where $\sigma \in \mathbf{\Sigma}$ denotes a system state, we denote by

[9] We shall come across similar notation (i.e., $osc_{\mathbf{L},\emptyset,\mathbf{H}}$) later when discussing unwinding conditions

$\sigma' \in \mathbf{\Sigma}$ the next state after a transition from σ, and write $reachable(\sigma)$ to mean that the system is able to reach state σ. Using *set difference* notation, we write $\mathbf{E}\backslash\mathbf{H}$ to denote the set of events in $\mathbf{E}$ with events in $\mathbf{H}$ removed. And for consistency, we write $\approx_L \subseteq \mathbf{\Sigma} \times \mathbf{\Sigma}$ as a binary operation describing *an equivalence relation* between states to denote the notion that both states are indistinguishable from the low security user's view of the system. We now present the two unwinding conditions defined by [99] as follows:

Definition 7 (Unwinding Condition 1)**.** *Any execution of a high security event in a reachable system state of a SES resulting in a new state should not be detectable by a low security user's observation of both the initial and final states of the system, i.e., l* **locally respects H** *[100], [99].*

This first unwinding condition (Definition 7) could be represented formally as given in Formula 2.6 below, where we use Mantel's notation lr_H, abbreviation for '*l locally respects H*', to denote the unwinding condition:

$$\begin{aligned} lr_H \triangleq\ & \forall(\sigma_1, \sigma_2) \in \Sigma, \forall h \in \mathbf{H}, \\ & ((reachable(\sigma_1) \wedge \langle \sigma_1, h, \sigma_2 \rangle \in \mathbf{T_R}) \Rightarrow \sigma_1 \approx_L \sigma_2) \end{aligned} \tag{2.6}$$

The second unwinding condition defined by Mantel is particularly interesting as it can be seen that it mirrors the semantics of the *classical refinement relation*. Notice this in Definition 8 below and the graphical representation in figure 2.7.

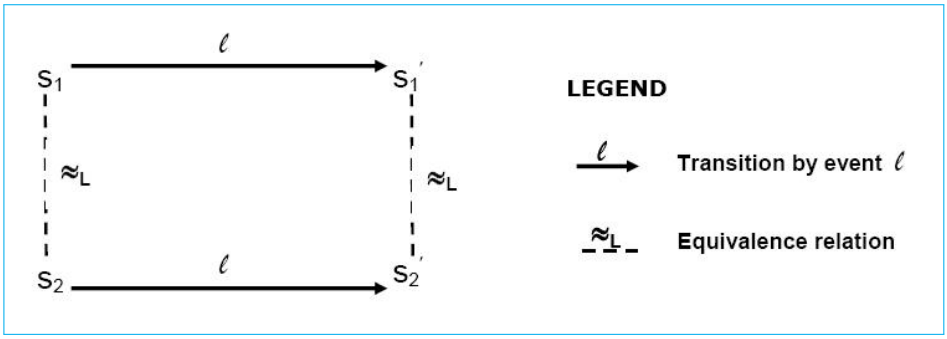


Figure 2.7: Output and Step Consistent (osc) Condition illustration

Definition 8 (Unwinding Condition 2)**.** *If two abstract states, σ_1 and σ_2, of a state event system are related by the relation '$\approx_L$' and the system is*

transitioned from σ_1, to another state σ_1', then there must exist a possible implementation σ_2' such that σ_2' simulates σ_2 and the relation '$\approx_L$' holds between σ_1' and σ_2'.

Formally, this condition could be represented, using another of Mantel's notation $osc_{\mathbf{L},\emptyset,\mathbf{H}}$ (coined from the phrase **o***utput and* **s***tep* **c***onsistent*), as shown in Formula 2.7 below.

$$\begin{aligned} osc_{\mathbf{L},\emptyset,\mathbf{H}} \triangleq\ & \forall(\sigma_1,\sigma_1',\sigma_2) \in \mathbf{\Sigma}, \forall l \in \mathbf{E}\backslash\mathbf{H}\cdot \\ & (\sigma_1 \approx_L \sigma_2\ \wedge\ \langle\sigma_1, l, \sigma_1'\rangle \in \mathbf{T_R}) \Rightarrow \\ & (\exists\sigma_2' \in \mathbf{\Sigma}\cdot\langle\sigma_2, l, \sigma_2'\rangle \in T_R\ \wedge\ \sigma_1' \approx_L \sigma_2') \end{aligned} \tag{2.7}$$

Formula 2.7 basically shows that if a low security event occurs in some state, then it must also occur in all low equivalent states ($\approx_L$) for the system to be *output and step consistent* with the defined security policy. Subsequently, [99] introduced two refinement operators that could be used to preserve information flow properties through refinement. We discuss these refinement operators, their benefits and limitations in Section(2.2.3).

2.2.2.9 Equivalence Relations based Noninterference (ERN)

In this section we present a number of confidentiality properties that are based essentially on the notion of equivalence relations on sets of system behaviour with respect to the low security observer's view of system properties via permitted system behaviour.

Bisimulation-Based Nondeducibility on Composition (BNDC)
The notion of Bisimulation-based Nondeducibility on Composition (BNDC) was introduced by Focardi and Gorrieri [57] as a characterisation of Noninterference in the *Security Process Algebra*[10] (SPA) calculus. The basic notion of BNDC is that a system C is BNDC if a low security user's view of the system is not modified when the system is composed with any high security process. The key intuition is to detect and prevent apparently innocent high security processes (*Trojan horse processes*) from maliciously leaking secure information to a public observer.

[10]Security Process Algebra: A variation of Milner's Calculus of Communicating Systems (CCS) [106] used to specify multilevel secure systems by partitioning the set of *visible* actions into *high* and *low* level actions.

There are two primary strains of BNDC, namely: Strong BNDC (SBNDC) and Persistent BNDC (P_BNDC). SBNDC is an information-flow security property that requires, for all reachable states, that the low security user's view of the system before a high security action is executed in composition with a low security action must be the same as after the high security event is executed. P_BNDC goes one step further by requiring the BNDC property for processes in dynamic contexts, i.e., contexts that can be reconfigured at runtime [29]. Both types of BNDC are compositional with respect to the parallel operator, but it is well known that they are not compositional with respect to the Nondeterministic choice operator. Hence they cannot be used to specify systems designed using the B Method where the Nondeterministic choice operator is a main feature. To overcome this limitation, Bossi et al [28] introduced an extension to P_BNDC, which they termed Compositional P_BNDC (CP_BNDC). The primary benefit of CP_BNDC is that it is fully compositional in that it is compositional, not only with respect to the parallel operator, but also with respect to the Nondeterministic choice operator.

For consistency with our work, and to make the notion of BNDC more understandable, we present the syntax and semantics of SPA, using the operational semantics of '*Labeled Transition System*' (LTS) using language-based syntax rather than trace-based process algebra representation. This would also make it easier to overcome the problem of avoiding the universal quantification on high level processes in the SPA characterisation of BNDC [28]. Further we present the characterisation of BNDC using bisimulation-like equivalence relations.

Given that C denotes a process; Z denotes a constant that must be associated with a definition $Z \triangleq C$. We use ξ to denote the set of all processes, and as SPA requires, ξ is partitioned into $\xi_{\mathbf{H}}$ and $\xi_{\mathbf{L}}$ respectively denoting sets of *high level processes* and *low level processes*. $C\backslash H$ means process C executed with all high security sub-processes removed. We use $\mathcal{A}$ to denote the set of all actions (both internal and visible actions), and while we use a to range over $\mathcal{A}$, we employ v to range over the set of *visible* actions, which we denote as $\mathcal{V}$. Given that **I** stands for the set of input actions while **O**

Syntax	Description
$C ::= skip$	Empty process
$\mid Z$	Constant
$\mid$ **SELECT** a **THEN** C **END**	Prefix
$\mid C_1[]C_2$	Nondeterministic Choice
$\mid C_1 \parallel C_2$	Parallel Composition
$\mid \xi\backslash\upsilon$	Restriction on visible action
$\mid \xi[f]$	Relabeling

Table 2.3: SPA Syntax, using Language based characterisation

stands for the set of output actions, then clearly the set of visible actions, $\mathcal{V} = \mathbf{I} \cup \mathbf{O}$. We write $\upsilon \in \mathcal{V}$ to range over the set of visible actions. Using τ to denote *internal* actions, it follows that $\mathcal{A} = \mathcal{V} \cup \{\tau\}$. To distinguish between input and output actions, we add an *overline* to the character to denote an output action, e.g., we write $a, b, \ldots$ to mean input actions and $\overline{a}, \overline{b}, \ldots$ to mean output actions. We define a function on the set of actions as $f : \mathcal{A} \to \mathcal{A}$, and represent the notion of the possibility of zero or more transitions as $(\to)*$. Finally, we write $\stackrel{a}{\Longrightarrow}$ to denote the notion that a sequence of zero or more transitions terminate in a *reachable* state. With this notation, we present the SPA syntax in Table 2.3, and a definition of *Weak Bisimulation* with which BNDC is developed in Definition 9.

Definition 9 (Weak Bisimulation)**.** *For all $a \in \mathcal{A}$, a relation $\mathcal{R} \subseteq \xi \times \xi$ over processes is a weak bisimulation, denoted $\approx_w$, if $(C_1, C_2) \in \mathcal{R}$ implies:*

1). If $C_1 \stackrel{a}{\to} C_1'$, then $\exists C_2' \cdot C_2 \stackrel{a}{\Longrightarrow} C_2' \wedge (C_1', C_2') \in \mathcal{R}$;
2). If $C_2 \stackrel{a}{\to} C_2'$, then $\exists C_1' \cdot C_1 \stackrel{a}{\Longrightarrow} C_1' \wedge (C_1', C_2') \in \mathcal{R}$;

We can now define BNDC as follows:

Definition 10 (BNDC)**.** *Given that $C \in \xi$, and C_h is a high security process,*

$$C \in BNDC \iff \forall C_h \in \xi_{\mathbf{H}} \cdot C\backslash H \approx_w (C \parallel C_h)\backslash H$$

Definition 10 basically states that whenever a low security observer cannot distinguish whether a process is executed in composition (parallel only) with

a high security process or not, for all possible high security processes, such a system is said to be BNDC. Following from this, we follow through with the definitions of P_BNDC and SBNDC in the literature.

The notion of P_BNDC was introduced by Focardi and Rossi [58] as a means of analysing systems in dynamic environments, based on the intuition that a system C is P_BNDC if it never reaches insecure states - hence the term '*persistent*' is based on the notion that all reachable states are *always* secure. Semi-formally, Given that $C \in \xi$, we write:

$$C \in P\_BNDC \iff \forall\, reachable(C')\ from\ C,\ C' \in BNDC \tag{2.8}$$

To avoid the problem of the universal quantification in the semi-formal Formula 2.8, [28] defined P_BNDC as a *weak bisimulation up to H*. Writing $\overset{a}{\Longrightarrow}_{\backslash H}$ to denote a generalisation of the terminating transition relation $\overset{a}{\Longrightarrow}$ on the set of low security observable actions, whereby:

$$\overset{a}{\Longrightarrow}_{\backslash H} = \begin{cases} \overset{a}{\Longrightarrow} & \text{if } a \notin \mathbf{H} \\ \overset{a}{\Longrightarrow} \text{ or } \overset{\tau}{\Longrightarrow} & \text{if } a \in \mathbf{H} \end{cases} \tag{2.9}$$

we give the authors' characterisation of weak bisimulation up to H as follows.

Definition 11 (Weak Bisimulation up to H)**.** *A binary relation $\mathcal{R} \subseteq \xi \times \xi$ over processes is a weak bisimulation up to H [106] if $\forall a \in \mathcal{A}$, $(C_1, C_2) \in \mathcal{R}$ implies that:*

1). If $C_1 \overset{a}{\rightarrow} C_1'$, then $\exists C_2' \cdot C_2 \overset{a}{\Longrightarrow}_{\backslash H} C_2' \wedge (C_1', C_2') \in \mathcal{R}$;
2). If $C_2 \overset{a}{\rightarrow} C_2'$, then $\exists C_1' \cdot C_1 \overset{a}{\Longrightarrow}_{\backslash H} C_1' \wedge (C_1', C_2') \in \mathcal{R}$;

Given that $\approx_{\backslash H}$ stands for the relation 'weak bisimulation up to H', two processes C_1, C_2 are thus related, written $C_1 \approx_{\backslash H} C_2$, if $\forall a \in \mathcal{A}, (C_1, C_2) \in \mathcal{R}$. Hence:

$$C \in P\_BNDC \iff C \approx_{\backslash H} C\backslash H$$

Focardi and Rossi [58] pointed out that the notion of SBNDC introduced by [57] is subsumed by their formulation of P_BNDC. i.e.,

$$SBNDC \subseteq P\_BNDC \subseteq BNDC.$$

SBNDC requires that the low security observer's view of the system appear to be the same both before and after the execution of a high security action. This implies that every state reachable from a secure system is itself secure, and this notion is defined as a '*weak bisimulation up to H with no internal action*', since termination is assumed. Given that $\approx^0_{\backslash H}$ denotes weak bisimulation up to H with no internal action, this equivalence relation is defined as follows.

Definition 12 (Weak Bisimulation up to H with no τ)**.** *A binary relation $\mathcal{R} \subseteq \xi \times \xi$ over processes is a weak bisimulation up to H with no internal action if $\forall a \in \mathcal{A}$, $(C_1, C_2) \in \mathcal{R}$ implies that:*

1). If $C_1 \stackrel{a}{\rightarrow} C_1'$, then $\exists C_2' \cdot C_2 \stackrel{a}{\Longrightarrow}^0_{\backslash H} C_2' \wedge (C_1', C_2') \in \mathcal{R}$;
2). If $C_2 \stackrel{a}{\rightarrow} C_2'$, then $\exists C_1' \cdot C_1 \stackrel{a}{\Longrightarrow}^0_{\backslash H} C_1' \wedge (C_1', C_2') \in \mathcal{R}$;

Thus

$$C \in SBNDC \iff C \approx^0_{\backslash H} C\backslash H$$

To overcome the limited compositionality of earlier strains of BNDC, Bossi et al [28] introduced the notion of *Compositional BNDC*, abbreviated CP_BNDC. CP_BNDC is fully compositional in the sense that it is compositional with respect both to the parallel composition operator and the Nondeterministic choice operator. This derives from the definition of a new observational equivalence termed, *weak bisimulation up to H with at least one τ*. This allows actions from H to be matched by one or more internal actions, τ, unlike the case of SBNDC where such internal actions are not allowed. Using $\Longrightarrow^+_{\backslash H}$ to denote this equivalence relation, the definition of CP_BNDC is given below:

Definition 13 (CP_BNDC)**.** *Given that $C \in \xi$, CP_BNDC can be characterised as a bisimulation in terms of $\approx^+_{\backslash H}$, namely:*

$$C \in CP\_BNDC \iff C \approx^{+}_{\backslash H} C\backslash H$$

Following from the definitions of P_BNDC and CP_BNDC, it can be seen that while there is no direct relationship between SBNDC and CP_BNDC, CP_BNDC is a special variant of P_BNDC. Hence, the authors noted the following relationship between P_BNDC and CP_BNDC.

$$CP\_BNDC \subseteq P\_BNDC \subseteq BNDC$$

A generalisation of security properties introduced by Alur et al [6], parameterised on equivalence relations and somewhat similar to BNDC is discussed in Section 2.3.5.4, *Inferable Properties based Secrecy Refinement*. In the meantime, we continue with our literature review as we discuss some language-based Noninterference properties in Section 2.2.2.10 below.

2.2.2.10 Language-Based Confidentiality Properties

In this section we present a number of confidentiality properties in the literature that we term language-based confidentiality properties for the reason that the authors employed programming language concepts in their formalisation of secrecy. We begin with Bicarregui's framework based on *read* and *write* frames.

Read/Write Frame based Confidentiality Property

Bicarregui in [23] proposed a way of extending the B Method's Generalised Substitutions to incorporate the semantics of read and write frames in model-oriented specification in order to preserve the confidentiality of secret state variables. In this context, the read frame of a program statement (or substitution) is the set of all program variables *read* on execution of the substitution in a specified state. Conversely, the write frame is the set of all variables *written* to during the execution of the substitution in a specified state. Given two program statements (Generalised Substitutions) S_1 and S_2, let '$\|$' denote *parallel composition* of the substitutions, ';' stands for *sequential composition* of the substitutions, and let '$\sqsubseteq$' denote "*is refined by* ". On the relevance of read frames, the author pointed out that a necessary condition for $S_1 \parallel S_2$ to be refined by $S_1; S_2$ is that S_2 does not depend on S_1, neither does it read any variable written by S_1, (i.e., '*the read*

frame of S_2' and '*the write frame of* S_1' are disjoint sets). Hence, writing $reads(S)$, $writes(S)$ to denote, respectively, the *read frame* and *write frame* of the Generalised substitution S, we have:

$$S_1 \parallel S_2 \sqsubseteq S_1 \ ; \ S_2 \ \subseteq \ reads(S_2) \ \cap \ writes(S_1) = \varnothing.$$

The author used an example involving substitution to a boolean variable x to show that, by the classical refinement semantics, x could be refined by $x := true$, $x := false$, $x := x$, $x := \neg x$, or for some other boolean variable y, $x := y$. He then suggested that by specifying a *do not read this* constraint on substitutions to x, such that x is not allowed to read any variable, one could prevent interference through substitutions like $x := y$. The author's main thrust in this paper is to partition the state space with the aid of *termination conditions* and *meaning relations* in such a way that only refinements that satisfy these constraints are accepted as valid. Thus, the resulting set of valid refinements is a subset of the refinements admitted by the classical refinement relation. This framework, however is not sufficient to guarantee Noninterference generally, especially in dealing with Nondeterministic substitutions and termination-sensitive Noninterference [23].

In formalising the '*Do Not Read This*' framework, [23] used Dijkstra's weakest precondition semantics. Thus we provide a fleeting introduction to the notion of weakest precondition semantics (wp) here. Given a program S, and a predicate Q, the weakest precondition under which S is guaranteed to establish the predicate Q as postcondition is the predicate that uniquely captures the largest set of initial states from which S can execute to establish Q. Continuing with the notation here, the weakest precondition is commonly denoted as $[S]Q$ or $wp(S, Q)$. It is worth pointing out here that wp requires that the program under consideration terminates. (A detailed discussion of wp will be presented later in Section 2.3.4.) It is understandable, therefore, that Bicarregui in [23] wrote $\neg[S]\neg(\sigma = \sigma')$ to denote the notion that: *given that* σ, σ' *are program states, it is not the case that the program* S *terminates in some state that does not satisfy the postcondition* $\sigma = \sigma'$.

The semantics of substitutions is enriched with notions of *read* and *write* frames. Given that $\mathbf{F}$ denotes the set of all variables in scope, $\mathbf{R} \subseteq \mathbf{F}$ is

the set of variables in the read frame, $\mathbf{W} \subseteq \mathbf{F}$ denotes the set of variables that can be written to, i.e., the write frame, and S is the body of the generalised substitution, a substitution is captured as a quadruple $(\mathbf{F}, \mathbf{R}, \mathbf{W}, S)$. A '*meaning relation*', $\mathcal{M}$, formalised as a predicate set on the states of a substitution, $\mathcal{S}$, is defined as: '*if the body of a substitution terminates successfully, then it must be the case that both the state before and the state after the substitution belong to the state space spanned by* $\mathbf{F}$ '. (Note that $\mathcal{M}$ corresponds to a relation on states, which yields the set of all possible state transitions [22]). Thus, if $\mathbf{\Sigma}$ denotes the state space spanned by $\mathbf{F}$, $\sigma \in \mathbf{\Sigma}, \sigma' \in \mathbf{\Sigma}$ ranges over the state before and the state after the substitution respectively, and using *weakest precondition* notation, the author defined $\mathcal{M}$ as follows:

$$\mathcal{M} \ \triangleq \ \{(\sigma, \sigma') \in \mathbf{\Sigma} \times \mathbf{\Sigma} \mid \neg[\mathbf{S}]\neg(\sigma = \sigma')\}$$

The author represented noninterference semantics in terms of read / write frames by devising a condition whereby the initial values of *write-only* variables variables may only influence their own values and those of other write-only variables, but may not influence the final values of *read* variables. Writing $\mathcal{M}_R$ to denote the meaning relation with respect to the read frame of program S, and given that $\sigma[S]_R$ denotes the the read frame of S in state σ, [23]'s noninterference condition could be represented as:

$$\forall \sigma_1, \sigma_2 \ \cdot \ \sigma_1 \ \mathcal{M}_R \ \sigma_2 \ \Rightarrow \ \sigma_1[S]_R = \sigma_2[S]_R$$

Clearly, from the foregoing, any final value of read variables possible from any start state is consistent with any other final value from any other start state. Hence, in this context, the substitution, S, does not read the values of variables in $\mathbf{F} - \mathbf{R}$, and given that $write_{F-R}$ denotes the least refined substitution which has $\mathbf{F} - \mathbf{R}$ as write frame, it follows that $write_{F-R}$ only affects variables in the set $\mathbf{F} - \mathbf{R}$, i.e., $write_{F-R}$ does not *interfere* with $\mathbf{R}$. Consequently, [23] reasoned that *whatever behaviour was possible for* S *from a given state, will also be possible if preceded by the execution of* $write_{F-R}$ [23]. This condition was then formalised as:

$$S; write_{F-R} \ \sqsubseteq \ write_{F-R}; S$$

Assuming the meaning of the substitution $write_{F-R}$ is $\mathbf{\Xi_R}$, and writing $readsM(\mathbf{R}, \mathcal{M})$ to denote the condition for not reading outside $\mathbf{R}$, the au-

thor summarised his characterisation of this nonintererence condition as:

$$readsM(\mathbf{R}, \mathcal{M}) \;\triangleq\; \Xi_R ; \mathcal{M} \subseteq \mathcal{M} ; \Xi_R$$

A termination condition describes the notion that the substitution respects the read frame and captures the condition under which the substitution is guaranteed to terminate [22]. This is formalised as a constraint on the relational image of a set $\mathcal{T}$ of variables, i.e., the set $\mathcal{T}$ is read frame respecting if it is a cylinder in the state space. Given that $(\!|\mathcal{T}|\!)$ denotes the relational image of $\mathcal{T}$, the read frame respecting termination condition, $readsT(\mathbf{R}, \mathcal{T})$, is formalised as:

$$readsT(\mathbf{R}, \mathcal{T}) \;\triangleq\; \mathbf{\Xi_R}(\!|\mathcal{T}|\!) \;\subseteq\; \mathcal{T}$$

The author then used the meaning relation along with the termination condition to define three compliance conditions for Noninterference, namely: *subst*, *writes*, and *reads*. These conditions are defined in terms of the $(\mathcal{T}, \mathcal{M})$ pair ranging over all valid refinements of the substitution, written as $(\mathcal{T}, \mathcal{M}) \in \mathcal{D}_o$, where $\mathcal{D}_o$ denotes the set of all valid refinements. The first condition, *subst*, ensures the substitution terminates with a read respecting meaning; *writes* ensures that the write frame is adhered to; and *reads* ensures that the read frame is respected.

These conditions could be expressed as:

$$subst_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T}, \mathcal{M}) \triangleq \mathcal{T} \supseteq [S]true \;\wedge\; \mathcal{M} \subseteq \neg[S]\neg(\sigma = \sigma') \tag{2.10}$$

$$writes_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T}, \mathcal{M}) \triangleq \mathcal{M} \subseteq \Xi_{F-W} \tag{2.11}$$

$$reads_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T}, \mathcal{M}) \triangleq \mathbf{\Xi_R} ; \mathcal{M} \subseteq \mathcal{M} ; \mathbf{\Xi_R} \;\wedge\; \mathbf{\Xi_R}(\!|\mathcal{T}|\!) \subseteq \mathcal{T} \tag{2.12}$$

And the set of all interpretations satisfying the conditions is given as:

$$\mathcal{S} \;=\; \left\{ (\mathcal{T}, \mathcal{M}) \in \mathcal{D}_\mathbf{o} \;\middle|\; \begin{array}{r} subst_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T}, \mathcal{M}) \\ \wedge\; writes_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T}, \mathcal{M}) \\ \wedge\; reads_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T}, \mathcal{M}) \end{array} \right\} \tag{2.13}$$

Notice these conditions all depend on the $(\mathcal{T}, \mathcal{M})$ pair. Consequently, the approach in [23] requires that termination is guaranteed. Although this would suffice for many practical systems, this limitation could be exploited

by a *strong adversary* to *implicitly* leak information through a substitution's termination behaviour. For example, consider the substitution below, where h holds a secret and l is a public variable, and *loop* is a substitution that never terminates.

$$\mathcal{S} \triangleq \textbf{if } h = l \textbf{ then } loop \textbf{ else } skip \textbf{ end}$$

Clearly this substitution may or may not terminate (i.e., termination is not guaranteed), hence the first part of the conjunct in $subst_{(\mathbf{F},\mathbf{R},\mathbf{W},S)}(\mathcal{T},\mathcal{M})$ is not guaranteed to hold. Consequently, an adversary could deduce due to this *implicit* information flow resulting from termination behaviour whether $h = l$ or not.

Using the notions of read-respecting substitution described in [23], Bicarregui illustrated how the problem of badly formed substitutions such as $x := x$ could be filtered out of possible refinements of boolean substitutions. Given that $x, y \in \{0,1\}$, and ...

$$(\mathbf{F},\mathbf{R},\mathbf{W},S) \equiv (\{x,y\},\ \{x\},\ \{y\},\ y := 0 [] y := 1)$$

The most general $(\mathcal{T},\mathcal{M})$ satisfying the read-respecting refinement conditions are calculated to be:

Termination set for $\{x,y\}$, $\mathcal{T} = \{(0,0),(0,1),(1,0),(1,1)\}$
And the meaning relation of S is:

$$\{(0,0) \mapsto (0,0),\ (0,0) \mapsto (0,1),\ (0,1) \mapsto (0,1),$$
$$(0,1) \mapsto (0,0),\ (1,0) \mapsto (1,0),\ (1,0) \mapsto (1,1),$$
$$(1,1) \mapsto (1,1),\ (1,1) \mapsto (1,0)\}$$

Notice that the value of x remains unchanged in all transitions of the system, hence the read frame constraint for $\{x\}$ is respected, and other refinements possible by the classical refinement relation, particularly refinements like $y := x$, $y := y$ or $y := 1 - y$, are not admissible. Our flow-logic based framework discussed in Section 3.6 reveals some interesting parallels between our work and Bicarregui's read and write frame based framework [23].

Next up for review is the semantics based information flow security intro-

duced by K. Rustan M. Leino and Rajeev Joshi[80], which we here term '*program equality property*'.

Program Equality Property (PEP)

Joshi and Leino in [80] presented a semantic characterisation of information flow security in a system operating on *public* and *private* variables. Their framework seeks to ensure that observations of the public variables before and after execution do not reveal any information about the initial values of the private variables. Hence they introduced a definition of security based on *program equality* (i.e, equality of two program terms with respect to the set of possible observations of low security observers). In this definition, HH stands for the program: "assign an arbitrary value to h", which the authors termed '*harvoc on h* '; S denotes a program statement; $\doteq$ stands for program equality, meaning that the LHS[11] and RHS[12] of $\doteq$ are equal (with respect to the set of possible low-security observations of both programs), provided the program S does not depend on the initial value of h, which is the variable holding the secret. ';' denotes sequential composition. Using this notation, a program is defined as secure if Formula 2.14 holds.

$$S \; is \; secure \; \triangleq \; (HH; S; HH \doteq S; HH) \tag{2.14}$$

The LHS of Formula 2.14 means that an arbitrary value is assigned to h before S is executed, and the final value of h is thereafter discarded. The RHS, on the other hand executes S, and then discards the final value of h resulting from execution of S by doing 'harvoc on h'. In both cases, the observer is unable to deduce the initial value of h by observing the initial and final values of the low security variables. Note that assigning HH after execution of S in both cases indicates that only the final value of the low level variables, not h, is required. This in itself could limit the usability of the authors' framework. For example, where a program needs to output the final value of h, the HH requirement will overwrite such high security values. However, one major advantage of the authors' framework is that it is not tied to any particular program syntax or semantics, hence it can be used to reason about any programming construct with clearly defined

[11]LHS - Left Hand Side of an operation
[12]RHS - Right Hand Side of an operation

semantics.

The authors used relational semantics to justify Formula 2.14 and to show how their framework relates to other notions of secure information flow in the literature. The notation used include $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ to denote program states; h, k denotes 'high-security variable' and 'low-security variable' respectively; $\sigma_1.h$ and $\sigma_1.k$ respectively denotes the values of h at σ_1 and k at σ_1. A relation $\mathcal{R}$ that relates an initial state σ_1 to a final state σ_1' is written as: $\sigma_1\langle\mathcal{R}\rangle\sigma_1'$. The identity relation denoted Id is defined for all σ_1 and σ_2 as: $\sigma_1\langle Id\rangle\sigma_2 \Leftrightarrow \sigma_1 = \sigma_2$.

Hence, the relational semantics of HH could be expressed as

$$\forall(\sigma_1, \sigma_2),\ \sigma_1\langle HH\rangle\sigma_2 \ \triangleq\ \sigma_1.k = \sigma_2.k \tag{2.15}$$

Formula 2.15 above basically shows that whatever an adversary knows about the initial value of h at state σ_2 (i.e., the final value of h) by observing both the initial and final values of k (i.e., $\sigma_1.k$ and $\sigma_2.k$) is no more than what he knew about h at the initial state σ_1. Subsequently, writing $(\forall j : rj : tj)$ to denote 'for all values of j satisfying the *range* $r.j$, the term or (*statement*) $t.j$ *holds*', and $\{j : r.j : t.j\}$ to denote 'the set of all elements of the form $t.j$ defined on j ranging over $r.j$', the following derivation of the definition of secure information flow was developed in line with existing definitions in the literature.

$$\begin{aligned} &S \textit{ is secure } \triangleq \\ &(\forall\sigma_1, \sigma_2 : \sigma_1.k = \sigma_2.k : \{\sigma_3 : \sigma_1\langle S\rangle\sigma_3 : \sigma_3.k\} = \{\sigma_4 : \sigma_2\langle S\rangle\sigma_4 : \sigma_4.k\}) \end{aligned} \tag{2.16}$$

This security condition could be read to mean: 'given that the values of k in start states σ_1 and σ_2 are equal, then if the program S executes from σ_1 to get to state σ_3, it must be the case that there is a state σ_4 such that S executes from state σ_2 and gets to state σ_4, and the value of k in σ_3, (i.e., $\sigma_3.k$), is equal to the value of k in σ_4, (i.e., $\sigma_4.k$). Clearly, this corresponds to many of the *equivalence relations* based notions of secrecy in the literature [111].

We précis Dijkstra's weakest precondition (wp) semantics in the preceding subsection of Section 2.2.2.10. Since Joshi and Leino [80] used both the notions of weakest precondition semantics and weakest liberal precondition (wlp) semantics in their work, we present an abridged introduction to wlp here. The notion of weakest liberal precondition semantics is an extension of wp by the same author into the realms of partial correctness. Weakest liberal precondition is predicated on the largest set of initial values from which a program S, *if it terminates*, is guaranteed to satisfy the postcondition Q. wlp is commonly formally denoted as $wlp(S, Q)$.

The secure information flow definition could also be presented in Djikstra's weakest precondition logic [80]. Writing $wp.S.p$ to mean the '*weakest precondition*' under which the predicate p holds (i.e $wp.S$ always terminates in a state that satisfies p), and $wlp.S.p$ to denote the '*weakest liberal precondition*' under which, if S terminates, the predicate p is satisfied, [80] show that secure information flow definition on hypothetical programs S_1 and S_2 could be represented as

$$S_1 \doteq S_2 \equiv (\forall p :: \ [wlp.S_1.p \equiv wlp.S_2.p] \ \wedge \ [wp.S_1.true \equiv wp.S_2.true]) \quad (2.17)$$

Now, taking S_1 to be the LHS of Formula 2.14, i.e., $HH; S; HH$, and S_2 to be the RHS, i.e., $S; HH$, the secure information flow definition can be rewritten by simply substituting for S_1 and S_2 in Formula 2.17, giving:

$$\begin{aligned} &HH; S; HH \doteq S; HH \equiv \\ &(\forall p :: \ [wlp.(HH; S; HH).p \equiv wlp.(S; HH).p] \ \wedge \ [wp.(HH; S; HH).true \equiv \\ &\qquad\qquad wp.(S; HH).true]) \end{aligned} \quad (2.18)$$

Since the program HH assigns an arbitrary value to h, it follows that whatever the value of h in S, the predicate p must be satisfied. Hence Formula 2.18 could be simplified by replacing HH with $\forall h$ in all its occurrences, resulting in the form presented in Formula 2.19 below.

$$\begin{aligned} &HH; S; HH \doteq S; HH \equiv \\ &(\forall p :: [\ [\forall h : wlp.S.[\forall h : p]\ ] \equiv wlp.S.[\forall h : p]\ ]) \wedge [\ [\forall h : wp.S.true] \equiv \\ &\qquad\qquad wp.S.true] \end{aligned} \quad (2.19)$$

For simplicity, expressions of the form $[\ [\forall h : q] \equiv q]$, termed "h-cylinders" or cylinders, was denoted by Cyl. $[\ [\forall h : q] \equiv q]$ indicate that q satisfies Cyl in as much as the value of Cyl does not depend on h. This assertion is written as $q \in Cyl$. Thus, expressing Formula 2.19 in terms of cylinders the result becomes:

$$S\ is\ secure \equiv (\forall p : p \in Cyl : wlp.S.p \in Cyl) \wedge wp.S.true \in Cyl \tag{2.20}$$

Recall from notation stated earlier that $p \in Cyl$ is the range of the quantification over p, and $wlp.S.p \in Cyl$ is the term of the quantification. To make Formula 2.20 amenable to automated verification, [80] used the logic notions of conjunctive [13] and disjunctive [14] spans to reduce the quantified expression in the first part of the conjunct on the RHS. By so doing, the quantification over predicates in Formula 2.20 is reduced to a first-order logic characterisation of the program equality definition in terms of cylinders. Taking f to be a universally conjunctive predicate transformer, X to denote any set of predicates and $\mathcal{A}.X$ the conjunctive span over subsets of X, the following theorem was formulated by the authors:

$$(\forall p : p \in X : f.p \in Cyl) \equiv (\forall q : q \in \mathcal{A}.X : f.q \in Cyl) \tag{2.21}$$

The need for this rewriting is the intuition that it is easier to deal with predicates in first-order logic than high-order quantifiers. Also, expressing the predicate as universally conjunctive (or disjunctive) is tantamount to breaking a difficult problem into smaller ones that could be solved individually and then composed to provide a solution to the original problem. Extending the conjunctive and disjunctive span logic to the simplified case where the predicate transformer, $wp.S$, is known to be deterministic, given that K is the set of all possible values of k, the program equality definition was transformed to:

$$S\ is\ secure \equiv (\forall K,\ wp.S.(k = K) \in Cyl) \tag{2.22}$$

We will, in Section 3.6, show how our framework relates to Joshi and Leino's

[13] Conjunctive span - the set of predicates obtained by taking conjunctions over subsets of a set of predicates

[14] Disjunctive spans - the set of predicates obtained by taking disjunctions over subsets of a set of predicates

program equality property. In the meantime, we follow with a discussion of the system model (introduced by [50]) based on the mathematical notion of lattices.

Lattice Model Approach to Confidentiality (LM)

Denning and Denning [51], [50], building on the work of Bell and LaPadula [21], introduced a mathematical framework for *program certification* centered on a lattice structure derived from the security classes such as described in [21]. The authors, [51], presented this work as an *Information Flow Model*, FM, formulated in terms of a quintuple of a set of *logical storage objects*[15], denoted $LSO \subseteq \{a, b, c \dots\}$, a set of processes, denoted $P \subseteq \{p, q, r \dots\}$, a set of disjoint security classes, denoted $SC \subseteq \{A, B, C \dots\}$, and two binary operations on SC, namely: the *class combining operator*, denoted $\oplus$, and the *flow relation*, denoted $\rightarrow$.

The class combining operator specifies, for any pair of operand classes, the resulting class of a function, f, on values of the operand classes. And $\oplus$ is both commutative and associative. Given, for example, that $a, b, c \in LSO$ and that $\underline{a}, \underline{b}, \underline{c} \in SC$ such that $f(a) = \underline{a}$, $f(b) = \underline{b}$, and $f(c) = \underline{c}$. Then the security class of the assignment statement $d := a+b+c$ is given as $\underline{d} = (\underline{a} \oplus \underline{b}) \oplus \underline{c}$, or $\underline{d} = \underline{a} \oplus (\underline{b} \oplus \underline{c})$. Since SC is intended as a *universally bounded lattice*[16], the set SC is closed under $\oplus$. The last component of the tuple, the flow relation, is a relation between security classes defining the notion that the left operand flows into the right operand, i.e., $\rightarrow \in SC \times SC$. Hence, $A \rightarrow B$ means that information stored in a LSO associated with the security class A flows into LSOs associated with the security class B. For *consistency*[17], $\rightarrow$ is transitive, i.e., if $A \rightarrow B$ and $B \rightarrow C$, then both policies imply $A \rightarrow C$. With this brief introduction to the building blocks of the flow model introduced in [51], we present below the flow model:

$$FM = \langle LSO, P, SC, \oplus, \rightarrow \rangle$$

[15] Logical storage objects are logical information receptacles or references to the receptacles, e.g., files, segments, objects, program variables, even users, etc. This is akin to our notion of *entities* earlier in this chapter.

[16] A universally bounded lattice is a structure composed of a partially ordered set, a greatest lower bound, and a least upper bound.

[17] Consistency in this context requires that flows implied by a permissible flow should also be permitted by the flow relation [51].

The last three components of FM form a universally bounded lattice $\mathcal{L} \triangleq \langle SC, \rightarrow, \oplus \rangle$ partially ordered on $\rightarrow$. Thus, $\langle SC, \rightarrow, \oplus \rangle$ has the following properties, namely:

- SC is finite;
- $\langle SC, \rightarrow \rangle$ is a partially ordered set;
- $\{\exists L \in SC \mid (\forall A \in SC)\ L \rightarrow A\}$ is a lower bound of SC [48]. Dually, there also exists an upper bound, i.e., $\{U \in SC | (\forall A \in SC)\ A \rightarrow U\}$;
- The least upper bound operator on SC corresponds to the class combining operator, hence $\oplus$ is also used to denote the least upper bound of SC. This is because when two or more classes are combined, the classification of the resulting composition corresponds to the security class of the component with the least classification.

The authors noted that the lattice structure could be either *linear*, e.g., the type commonly used in government: *unclassified, confidential, secret,* and *top secret*, or it could be a composite, nonlinear ordering on a finite set of properties of interest, or a combination of both. We give a graphical illustration of a linear lattice below.

$$unclassified \longrightarrow confidential \longrightarrow secret \longrightarrow top\ secret$$

Given that fin, med, crm denotes 'financial', 'medical', and 'criminal' records respectively, we illustrate a nonlinear lattice defined on the set of *personnel* records, $X = \{fin, med, crm\}$, with $\varnothing$ and X being the *greatest lower bound*, and the *least upper bound* respectively in Figure 2.8. The authors coined the terms '*explicit*' and '*implicit*' flows, which we described in Section 2.2. A static program certification mechanism was then introduced and compared with *run-time* or dynamic enforcement mechanisms. The approach is to incorporate this mechanism into the analysis phase of a compiler for a high-level language. The disadvantage of this is that the analysis may need to be repeated to adapt it to another language. Our approach to overcome this limitation is to separate the flow analysis phase from the

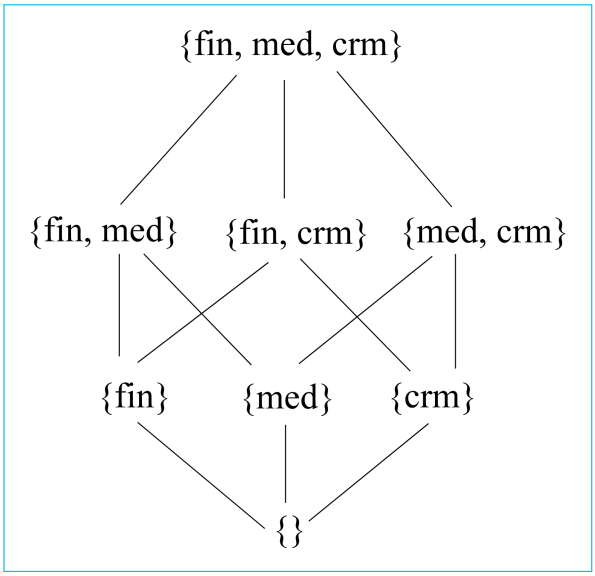


Figure 2.8: Non-Linear Lattice

program-specific analysis phase of a compiler, and from the problem of *refinement*, hence there is no need to *re-do* the analysis to adapt it to another language. The intuitions behind the certification mechanism are:

i. The security classification of an assignment statement is updated to the least upper bound, $\oplus$, of the security classes of the variables flowing into the assigned variable. For example, given that $a, b, c, d \in LSO$, $\underline{a}, \underline{b}, \underline{c}, \underline{d} \in SC$, $f(a) = \underline{a}, f(b) = \underline{b}, f(c) = \underline{c}, f(d) = \underline{d}$, and $S \triangleq d := a + b * c$, $\underline{d}$ is computed to be $\underline{d} = \underline{a} \oplus \underline{b} \oplus \underline{c}$. Hence, $\underline{d}$ is secure if and only if $(\underline{a} \oplus \underline{b} \oplus \underline{c}) \rightarrow \underline{d}$.

ii. The security classification of a *conditional* structure (e.g., an **IF**... **THEN** ...**ELSE** construct) is calculated to be the greatest lower bound, $\otimes$, of the variables in the structure. Thus, given the statement $S \triangleq$ **if** $c = 0$ **then** $a := 0$ **else** $b := 1$ **end**, the security class of the statement is computed to be $(\underline{a} \otimes \underline{b} \otimes \underline{c})$, and it must be the case that $\underline{c} \rightarrow (\underline{a} \oplus \underline{b} \oplus \underline{c})$.

Another limitation of the certification mechanism in [51] is that it does not take account of Nondeterminism, and it is not clear how to guarantee that valid refinements are secure on removal of Nondeterminism in the implementation. Further, although the examples provided are useful in helping the

reader understand the intuition behind the mechanism, no rigorous proofs of correctness were given. These problems we seek to address later in Section 3.4 of this thesis. We present in Table 2.4 a comparison of the static program certification mechanism in [51], [50], and the dynamic ones in the literature.

An extension of Denning and Denning's work to type systems was developed by by Volpano, Smith and Irvine. This is the next subject of our review.

Type-Based Approach to Confidentiality

Volpano et al in [76], [130] introduced a framework of type inference rules for making judgements about the flow of information within software systems developed in a procedural programming language. This work is a formulation of Denning's lattice-model approach [51] as a type system that can be regarded as having the Noninterference property, if soundness of the system is established. The authors developed their typing system based on the general intuition behind Noninterference, namely: 'a high security input can be modified without affecting the output visible to a low security observer.' The type system is fundamentally a three-layered system: the first layer constitute the *security levels* referred to as the τ types in a partially ordered set $\langle \tau, \leq \rangle$; the second are the *expression* types called the π types, and lastly the *phrase* types, denoted by ρ. The $\leq$ ordering is extended to a subtype relation $\subseteq$ over ρ. Variables that could be read are typed as τ *var* while output (or write-only) variables (including variables assigned to), termed *acceptors* are typed τ *acc*. Only variables with security classification less than or equal to τ are allowed to flow into any variable of type τ *var* or τ *acc*. A command type is of the form τ *cmd*, and intuitively, only assignments to variables with security classification greater than or equal to τ are allowed in τ *cmd*. Thus, the subtype relation is *contravariant* or antimonotonic with respect to command types, i.e., if $\tau \leq \tau'$, then τ' *cmd* $\subseteq \tau$ *cmd* [76]. We find this intuition easier to comprehend by means of our contrived geometric representations in Figure 2.9.

From Figure 2.9 we can see that whenever $\tau \leq \tau'$, then τ *cmd* could 'write up' to τ' *cmd*, hence with respect to what could be written, it is clear that *whatever security class* τ' *cmd could write to could also be written to by*

Certification Mechanisms	
Dynamic	**Static**
• Updating an object's class may remove the object from the low security user's view. This could cause implicit flows.	• Execution of program is guaranteed to be secure before hand. Not so with dynamic mechanisms.
• Impractical: many real world objects and most users often have fixed classification.	• Does not use up system resources at runtime since checks are done before execution.
• Often specified in terms of low-level hardware instructions	• Can be specified in terms of high-level language structures
• Run-time mechanisms may be able to trap 'bad behaviour' e.g. 'exceptions' in a program.	• Language implementation defects e.g., 'array out of bounds', 'division by 0', etc. could pose problems.
• Hardware malfunction does not impact mechanism, since certification is done during program run-time.	• Hardware malfunction during run-time could make statically certified programs insecure.
• Dynamic mechanisms are tied to the language in which it is written.	• Incorporating mechanism into the compiler analysis phase ties it *only* to the language of the development.
• Adversary may intentionally cause security violations to trigger implicit information leaks.	• Since certification is done before execution, intentional security violations cannot occur.

Table 2.4: Certification Mechanisms: Static v. Dynamic

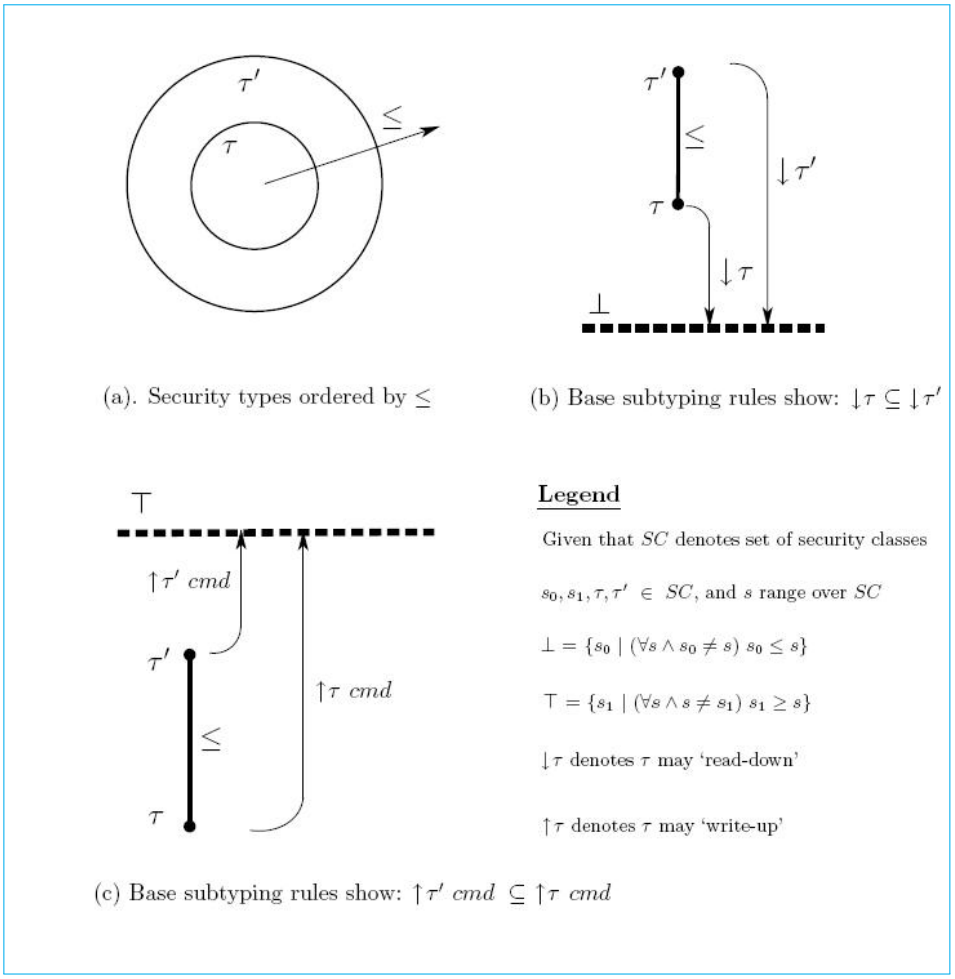


Figure 2.9: Geometry of security class ordering and subtyping

τ cmd, but not vice versa. Hence τ' *cmd* $\subseteq$ τ *cmd.*

Security typing judgments are made using two *finite* functions, namely: *identifier typing*, denoted γ, which maps identifiers to types of the form τ, τ *var*, τ *acc*; and *location typing*, written λ, which maps memory locations to τ types. Thus, the γ judgement $\gamma \vDash e : \tau$ *acc* means the phrase type e is an *acceptor* with security type τ, and $\gamma \vDash c : \tau$ *cmd* means the phrase type c is a *command* with security type τ. We could also write the above statements as $\gamma(e) = \tau$ *acc* or $\gamma(c) = \tau$ *cmd* respectively. Location typing follows a similar convention. The authors [76], [130] defined their type system on a core imperative language with the following notation and

syntax.

The variable ℓ range over memory locations; x range over identifiers; while n range over integer literals. e (primed, subscripted or *plain*) stands for expressions; p is a phrase; c denote commands; **letvar** denotes the allocation of a location in memory, with the deallocation shown by specifying the scope of the location using the keyword **in**; and similarly, **letproc** denotes the allocation and deallocation of a procedure. With this notation, the core imperative language and type structure is presented below.

$$
\begin{aligned}
(Phrase) \quad & p ::= e \mid c \\
(Expr) \quad & e ::= x \mid n \mid \ell \mid e + e' \mid e - e' \mid e = e' \mid \\
& e < e' \mid proc(\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c \\
(Comm) \quad & c ::= e := e' \mid c; c' \mid e(e_1,\ e_2,\ e_3) \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid \\
& \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' \mid \mathbf{letvar}\ x := e\ \mathbf{in}\ c \mid \\
& \mathbf{letproc}\ x(\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c\ \mathbf{in}\ c'
\end{aligned}
$$

Thus the three-level type structure employed in the core imperative language is given below, given that s ranges over security classes.

$$
\begin{aligned}
\tau &::= s \\
\pi &::= \tau \mid \tau\ proc(\tau_1,\ \tau_2\ var,\ \tau_3\ acc) \mid \tau\ cmd \\
\rho &::= \pi \mid \tau\ var \mid \tau\ acc
\end{aligned}
$$

The security typing system allows variable types to be changed or updated as needed. The notation $\gamma[x : \rho]$ means the security type of the identifier x is updated to ρ. Generally, judgements are written in the form $\lambda; \gamma \vDash p : \rho$. With the notation given thus far, the authors [130] formulated a number of security type inference (or judgement) rules, which we reproduce in appendix B. The authors then introduced two lemmata to capture the notions of *simple security* and *confinement* originally formulated by Bell and LaPadula [21].

Lemma 1 (Simple Security)**.** *If* $\lambda; \gamma \vDash e : \tau$*, then* $\forall \ell \in e,\ \lambda(\ell) \leq \tau$*, and for every* x *free in* e*,* $\gamma(x) \leq \tau$*.*

Lemma 2 (Confinement)**.** *If* $\lambda \vDash c : \tau\ cmd,\ \mu \vDash c \Rightarrow \mu',\ dom(\lambda) = dom(\mu)$*, and* ℓ *is a location assigned to in* c*, then* $\lambda(\ell) \geq \tau$ *or* $\mu(\ell) = \mu'(\ell)$

Finally, the authors [130] introduced some principal types designed to document all possible types of a program.

Flow-Sensitive Type-Based Approach to Confidentiality

The type-based approach to confidentiality exemplified by [76] is flow insensitive in the sense that the security types of variables remain 'fixed' throughout program execution. Referencing Hankin and the Nielsons [64], Hunt and Sands in [75] noted that a key difference between flow insensitivity and flow sensitivity is: given two distinct operations C_1, C_2, the result of analysing $C_1; C_2$ is the same as $C_2; C_1$ in flow insensitive frameworks whereas the results are different in flow sensitive analysis systems. That is, the order of execution is not taken into consideration in flow insensitive systems, thus all sub-programs *must* be secure for the analysis of the operation to be adjudged secure. Using similar intuition, Amtoft and Banerjee in [8] noted the imprecision of flow-insensitive type systems, showing that: given that l is a public variable and h a secret variable, such systems adjudge the program $l := h \ ; \ l := 0$ as insecure, a notion often termed *false positive* (i.e., judging a secure program as insecure). In reality, the final value of l does not depend on the initial value of h although the sub-program $l := h$ is insecure. Clearly, strict flow-insensitive type systems cannot distinguish between $l := h \ ; \ l := 0$ and $l := 0 \ ; \ l := h$. Flow-sensitive analysis frameworks, on the other hand, can distinguish between such programs, and will correctly judge the latter as *insecure* and the former as *secure*. Well-known flow-sensitive information flow analysis frameworks reviewed here include [8], [36], and [75].

Amtoft and Banerjee [8] formalised the notion of information flow in terms of *independence* of variables. Given that T denotes a set of traces, $[l \# h]$ denotes that the current value of variable l is independent of the initial value of the variable h, $T^{\#}$ denotes a set of independencies, the logic language introduced by [8] was defined on a finite abstraction of a (possibly infinite) set of independencies with subset inclusion. Intuitively, l is independent of h iff, for constant values of l in two sets of traces, any change in the initial value(s) of h in either trace, does not change the current value of l. The authors defined a Hoare-type logic of the form $\{T^{\#}\}\ C\ \{T_1^{\#}\}$. That is, given the precondition $T^{\#}$, and assuming termination of program C, C satisfies the postcondition $T_1^{\#}$. Given that **Ide** denotes an infinite

set of variables, an independence is defined as a set of $[l\#h]$ pairs, i.e., $T^{\#} \in \mathbf{independ} = \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$. The authors write **Cmd** to denote commands, with alternation and loop conditions termed **Context**, since the assignments / updates within such commands can only be made in the context of the control conditions.

A trace $t \in \mathbf{Trc}$ associates each variable with its initial value and its current value, i.e., $t : \mathbf{Ide} \to \mathbf{Val} \times \mathbf{Val}$. Given two traces t_1, t_2; and writing $\overset{x}{=}$ to denote that the current values of the variable x are equal in both t_1 and t_2, and writing $\overset{=}{x}$ to mean that the initial values of all variables other than x are equal in t_1 and t_2, it is adjudged that

$$[x\#y] \vDash T \;\Leftrightarrow\; \forall t_1, t_2 \in T \cdot t_1 \overset{=}{y} t_2 \;\Rightarrow\; t_1 \overset{x}{=} t_2, \qquad x \neq y \text{ and}$$
$$T^{\#} \vDash T \;\Leftrightarrow\; \forall [x\#y] \in T^{\#} \cdot [x\#y] \vDash T$$

Using trace semantics, where $[\![C]\!](T)$ denotes the set of traces resulting from the execution of command C from initial set T (or in the case of expression E, $[\![E]\!](T)$ denotes the set of traces resulting from the evaluation of E, given the initial values of the *free variables* in E (written $FV(E)$)), [8] assumed the existence of a function $[\![E]\!] : \mathbf{Trc} \to \mathbf{Val}$ which satisfies the property:

$$\forall x \in FV(E) \cdot t_1 \overset{x}{=} t_2 \;\Rightarrow\; [\![E]\!](t_1) = [\![E]\!](t_2)$$

Independencies are then computed using the function

$$sp : \mathbf{Context} \times\ \mathbf{Cmd} \times \mathbf{Independ} \to \mathbf{Independ}$$

Thus given $G \in \mathbf{Context},\ C \in \mathbf{Cmd}, T^{\#} \in \mathbf{Independ},\ sp(G, C, T^{\#})$ yields a postcondition $T_1^{\#}$ such that $G \vdash \{T^{\#}\}C\{T_1^{\#}\}$ holds and $T_1^{\#}$ is the largest set that makes the judgement hold [8]. That is, $T_1^{\#}$ is the largest set such that if $T^{\#} \vDash T$ then $T_1^{\#} \vDash [\![C]\!](T)$.

Using the Amtoft-Banerjee flow independence analysis framework, we extend the example in [8] to illustrate the fact that flow sensitive type systems distinguish between programs such as $l := h; l := 0$ and $l := 0; l := h$, assuming in both cases that $T_0^{\#}$, the set of initial independencies, is $\{[l\#h], [h\#l]\}$.

$\underline{C_1 \triangleq l := h; l := 0:}$

$$\{\ \{[l\#h],[h\#l]\}\ \}\ l := h\ \{\ \{[h\#l],[l\#l]\}\ \} \quad \text{and}$$
$$\{\ \{[h\#l],[l\#l]\}\ \}\ l := 0\ \{\ \{[h\#l],[l\#l],l\#h]\}\ \}$$
$$\therefore \{\ \{[l\#h],[h\#l]\}\ \}\ l := h; l := 0\ \{\ \{[h\#l],[l\#l],[l\#h]\}\ \} \quad \text{holds.}$$

Writing $T_1^{\#}$ for $\{[h\#l],[l\#l],[l\#h]\}$, we can see from the results above that $sp(\varnothing, C_1, T_0^{\#})$ computes $T_1^{\#}$ and $T_0^{\#} \subseteq T_1^{\#}$, hence $l := h; l := 0$ is adjudged *secure*.

$\underline{C_2 \triangleq l := 0; l := h:}$

$$\{\ \{[l\#h],[h\#l]\}\ \}\ l := 0\ \{\ \{[l\#h],[h\#l],[l\#l]\}\ \} \quad \text{and}$$
$$\{\ \{[l\#h],[h\#l],[l\#l]\}\ \}\ l := h\ \{\ \{[h\#l],[l\#l]\}\ \}$$
$$\therefore \{\ \{[l\#h],[h\#l]\}\ \}\ l := 0; l := h\ \{\ \{[h\#l],[l\#l]\}\ \} \quad \text{fails.}$$

Writing $T_2^{\#}$ for the final postcondition, $\{[h\#l],[l\#l]\}$, we can see from the results above that $sp(\varnothing, C_2, T_0^{\#})$ computes $T_2^{\#}$ and $T_0^{\#} \notin T_2^{\#}$, hence $l := 0; l := h$ is adjudged *insecure*.

For iterative commands [8] employs fixed point computation to compute $sp(G, C, T^{\#})$, in much the same way as Clark et al in [36].

The flow independence analysis framework introduced by [8] has the advantage of being able to judge more programs secure than flow-insensitive type systems such as the one introduced by Volpano et al [76]. Also any program that is well-typed in a Volpano et al system can also be judged secure in the Amtoft-Banerjee framework. As acknowledged by the authors, however, their approach, similar to Clark et al's though it is, is not *termination sensitive*. Thus, it will only work on the assumption the adversary cannot observe nontermination. This is one key advantage the Clark et al framework has over the Amtoft-Banerjee system.

Clark et al in [36] introduced a flow-sensitive and termination sensitive information flow analysis framework for a simple core imperative and Algol-like language, using a flow logic approach similar to the Hankin-Nielsons approach in [64]. The syntactic and semantic definitions of the categories within the core imperative language is also similar to the one pioneered by

Nielson and Nielson in [115], [114]. For example, the notion of *Free Variables*, of an expression E, denoted $FV(E)$ in [36], is akin to that defined in [115] where it is defined as the set of variables occurring in E, inductively defined formally as follows, where n denotes a natural number, x is a variable, and E_1, E_2 are expressions.

$$\begin{aligned}
FV(n) &= \varnothing \\
FV(x) &= \{x\} \\
FV(E_1 + E_2) &= FV(E_1) \cup FV(E_2) \\
FV(E_1 - E_2) &= FV(E_1) \cup FV(E_2) \\
FV(E_1 * E_2) &= FV(E_1) \cup FV(E_2) \\
FV(E_1/E_2) &= FV(E_1) \cup FV(E_2)
\end{aligned}$$

Since confidentiality properties often deal with the relationship between the initial values of secret variables and the final values of public variables, [36] used big-step (or natural) semantics in formalising the semantics of their core imperative language. Thus, for each statement in their language, the authors specified the relationship between the initial state and the final state, using the following notation. A program state, i.e., a set functions mapping each variable to a value, is denoted $\sigma, \sigma', \sigma''$, etc., a statement is represented as $S, S_1, S_2, \ldots, S_n$, the notion that a statement execution terminates is denoted by $\Downarrow$, and writing '*void*' after a program terminates indicate that there is nothing left to be executed after that particular statement terminates. This corresponds to the *null* or *empty* statement. Given that v is an integer value, the notion of variable update, written $\sigma[x \mapsto v]$ denotes that *only* the variable x is updated in state σ, and x thereafter has the value v. The value of a variable x in state σ is written as σx. A statement configuration, written S, σ indicates that the statement S is in state σ. 0, 1 denotes the boolean values $False$, $True$ respectively. Using a labeled transition system, where ℓ range over statement labels, [36] defined the semantics of their core imperative language as a set of inference rules, as shown below:

$$\frac{}{skip^{\ell}, \sigma \Downarrow void, \sigma}$$

$$\frac{E, \sigma \Downarrow v, \sigma}{(x := E)^{\ell}, \sigma \Downarrow void, \sigma[x \mapsto v]}$$

$$\frac{S_1, \sigma \Downarrow void, \sigma' \qquad S_2, \sigma' \Downarrow void, \sigma''}{(S_1; S_2)^\ell, \sigma \Downarrow void, \sigma''}$$

$$\frac{b, \sigma \Downarrow 1, \sigma \qquad S_1, \sigma \Downarrow void, \sigma'}{(if\ b\ then\ S_1\ else\ S_2)^\ell, \sigma \Downarrow void, \sigma'}$$

$$\frac{b, \sigma \Downarrow 0, \sigma \qquad S_2, \sigma \Downarrow void, \sigma'}{(if\ b\ then\ S_1\ else\ S_2)^\ell, \sigma \Downarrow void, \sigma'}$$

$$\frac{b, \sigma \Downarrow 0, \sigma}{(while\ b\ do\ S)^\ell, \sigma \Downarrow void, \sigma}$$

$$\frac{b, \sigma \Downarrow 1, \sigma \quad S, \sigma \Downarrow void, \sigma' \quad (while\ b\ do\ S)^\ell, \sigma' \Downarrow void, \sigma''}{(while\ b\ do\ S)^\ell, \sigma \Downarrow void, \sigma''}$$

$$\frac{S, \sigma[x \mapsto 0] \Downarrow void, \sigma'}{(new\ x.S)^\ell, \sigma \Downarrow void, \sigma'[x \mapsto \sigma x]}$$

Clark et al [36] defined three functions that were later used in the formulation of the security conditions for their information flow analysis framework. Given that **Ide** and **Lab** are restricted to the variables and labels appearing in the statement; $\widehat{X}$ defines the set of variables assigned to; $\widehat{D}$ defines the set of pairs of variables where the *first* variable in the pair depends on the *second*; and $\widehat{G}$ defines the set of variables that may affect the termination of the statement. The special metavariable $\bullet$ is used to denote that a loop has been encounterred, and hence there is the possibility of nontermination. The authors then denoted a set of variables, possibly containing a variable that may cause the program to fail to terminate by **Ide**, i.e., $\mathbf{Ide} = \mathbf{Ide} \cup \{\bullet\}$. The definition of the functions are given below:

$$\widehat{X} \in \mathbf{Assign} = \mathbf{Lab} \to \mathcal{P}(\mathbf{Ide})$$
$$\widehat{D} \in \mathbf{Dep} = \mathbf{Lab} \to \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$$
$$\widehat{G} \in \mathbf{Global} = \mathbf{Lab} \to \mathcal{P}(\mathbf{Ide})$$

Since the information flow analysis framework in [36] is extended and discussed in detail in Chapter 3 of this thesis, we skip the details here, and

present below the security conditions derived from the work, and an example showing how a simple *While* statement can be analysed down to the least fixpoint using the framework.

Security Conditions: Given that L, H denote sets of *low* and *high* security variables respectively. On analysis, a program C^{ℓ} is adjudged secure iff

- $H \cap \widehat{G}(\ell) = \varnothing$, and
- $\forall x \in L \cdot \nexists\, y \in H \cdot x\widehat{D}(\ell)y$

Fixpoint Analysis Example: Given the labeled program below where p, f are secret variables whereas q and x are public variables, we present the fixpoint analysis result using the Clark et al flow-logic framework in Table 2.5.

$$
\begin{array}{l}
(\ \textbf{while}\ x < 3\ \textbf{do} \\
\quad (\ \textbf{if}\ (p = q)\ \textbf{then} \\
\quad\quad (\ f := 1\ )^{\ell_6} \\
\quad\quad \textbf{else} \\
\quad\quad (\ f := 2\ )^{\ell_5}\ )^{\ell_2}; \\
\quad (\ (\ x := f + 1\ )^{\ell_4}; \\
\quad\quad (\ q := x - 1\ )^{\ell_3}\ )^{\ell_1} \\
\textbf{end}\ )^{\ell_0}
\end{array}
$$

Applying the security conditions given above to the analysis result presented in Table 2.5, it is clear that the program is not secure with respect to the given two-point security lattice wherefor variables are partitioned into sets of high and low security variables H and L respectively. The variables $p, f \in H$ are high security variables, and $q, x \in L$ are low security variables.

From table 2.5, $\widehat{G}(\ell_0) = \{f, p, q, x\}$
Thus $H \cap \widehat{G}(\ell_0) \equiv \{p, f\} \cap \{f, p, q, x\} \neq \varnothing$
$\therefore$ Flow is insecure (by the first security condition).

Similarly, by the second security condition:
$x \in L$ and $\widehat{D}(\ell_0) \supseteq x \mapsto f$, i.e., $x\widehat{D}(\ell_0)f$. But $f \in H$.
Thus $\forall x \in L \cdot \nexists\, f \in H \cdot x\widehat{D}(\ell_0)f$ fails
$\therefore$ Flow is insecure (by the second security condition).

FixpointComputation							
		Iteration1		**Iteration2**		**Iteration3**	
Labels	$\widehat{X}$	$\widehat{G}$	$\widehat{D}$	$\widehat{G}$	$\widehat{D}$	$\widehat{G}$	$\widehat{D}$
ℓ_0	$\{f,q,x\}$	$\{x\}$	$\{f \mapsto x, q \mapsto x, f \mapsto p, f \mapsto q, q \mapsto f, q \mapsto p, x \mapsto f\}$	$\{f, p, q, x\}$	$\{f \mapsto x, q \mapsto x, f \mapsto p, f \mapsto q, q \mapsto f, q \mapsto p, x \mapsto f, x \mapsto p, x \mapsto q\}$	$\{f,p, q,x\}$	$\{f \mapsto x, q \mapsto x, f \mapsto p, f \mapsto q, q \mapsto f, q \mapsto p, x \mapsto f, x \mapsto p, x \mapsto q\}$
ℓ_1	$\{q,x\}$	$\varnothing$	$\{q \mapsto f\}$	$\{x\}$	$\{q \mapsto f\}$	$\{f,p, q,x\}$	$\{q \mapsto f, q \mapsto p, q \mapsto x\}$
ℓ_2	$\{f\}$	$\varnothing$	$\{f \mapsto p, f \mapsto q\}$	$\{x\}$	$\{f \mapsto p, f \mapsto q, f \mapsto x\}$	$\{f,p, q,x\}$	$\{f \mapsto p, f \mapsto q, f \mapsto x\}$
ℓ_3	$\{q\}$	$\varnothing$	$Id[q \mapsto x]$	$\{x\}$	$\{q \mapsto x\}$	$\{f,p\ q,x\}$	$\{q \mapsto x, q \mapsto p\}$
ℓ_4	$\{x\}$	$\varnothing$	$Id[x \mapsto f]$	$\{x\}$	Id	$\{f,p\ q,x\}$	$\{x \mapsto p, x \mapsto q\}$
ℓ_5	$\{f\}$	$\varnothing$	$Id[f \mapsto \varnothing]$	$\{x\}$	$\{f \mapsto p, f \mapsto q, f \mapsto x\}$	$\{f,p\ q,x\}$	$\{f \mapsto p, f \mapsto q, f \mapsto x\}$
ℓ_6	$\{f\}$	$\varnothing$	$Id[f \mapsto \varnothing]$	$\{x\}$	$\{f \mapsto p, f \mapsto q, f \mapsto x\}$	$\{f,p\ q,x\}$	$\{f \mapsto p, f \mapsto q, f \mapsto x\}$

Table 2.5: (Flow) Fixpoint Computation of a simple while construct

Hunt and Sands in [75] introduced a novel dynamic security typing framework whereby the security types of variables are allowed to change (or '*float*') during computation, depending on the security type of information flowing into them. The authors follow the lattice-based approach introduced by Denning and Denning [51], and show that a universal type system could be built by using the universal lattice based on the flow lattice $\mathcal{P}(\mathbf{Ide})$, i.e., the powerset of the variables in the system. Hence they concluded that their universal type system subsumes other type systems in the literature since all possible typing in all possible lattices could be derived from one principal typing in the universal lattice [75]. They then show how their dependence-based approach could be transformed into the independence-based Amtoft-

Banerjee framework [8], or flow-insensitive types like Volpano, Smith and Irvine's [76].

Assuming **Ide** denotes a finite set of variables; $\mathcal{L}$ denotes a finite security lattice defined on $\mathcal{P}(\mathbf{Ide})$; type environments Γ, Γ' associate variables to the security lattice by means of the function $\Gamma, \Gamma' : \mathbf{Ide} \to \mathcal{L}$. Given a command C, and a security type $p \in \mathcal{L}$, secure flow judgements have the form $p \vdash_{\mathcal{L}} \Gamma\{C\}\Gamma'$. Intuitively, Γ records the security levels of variables in scope before execution of C, while Γ' records the security levels of those variables after C terminates, with the constraint that no variable with a security level lower that p is modified by C.

Hunt and Sands [75] used equivalence relations to define the noninterference condition that for all $t \in \Gamma$, and given the derivation $\Gamma\{C\}\Gamma'$, the final value of a variable x with type $t = \Gamma'(x)$ may depend only on those variables y with initial types $\Gamma(y) \sqsubseteq t$. This equivalence relation on stores (with respect to security type $t \in \Gamma$), denoted $=_{\Gamma,t}$, relates stores which are equal on all variables having type $u \in \Gamma$ where $u \sqsubseteq t$. Thus, given that σ, ρ denote stores, then

$$\sigma =_{\Gamma,t} \rho \;\Leftrightarrow\; \forall x \cdot \Gamma(x) \sqsubseteq t \Rightarrow \sigma(x) = \rho(x)$$

This equivalence relation is then used to define one of the requisite conditions for the semantic relation $p \vDash_{\mathcal{L}} \Gamma\{C\}\Gamma'$ to hold. These conditions, the first of which corresponds to the notion of '*Assignment Freedom*' in [36] are:

1. $\forall \sigma, \sigma', x \;\cdot\; \langle C, \sigma\rangle \Downarrow \sigma' \;,\; \Gamma'(x) \not\sqsupseteq p \Rightarrow \sigma'(x) = \sigma(x)$
2. $\forall t \in \mathcal{L},\; C : (=_{\Gamma,t}) \Rightarrow (=_{\Gamma',t})$.

The authors then introduced an algorithmic variant of their universal type system, which could be used in iterative fixpoint computation of the least fixed point of a monotome function on a finite lattice.

To facilitate a comparison of the universal type system with the Amtoft-Banerjee framework, Hunt and Sands in [75] translated both systems to a comparable notational system defined as mappings. $\Delta, \Delta' : \mathbf{Ide} \to \mathcal{P}(\mathbf{Ide})$ is used to range over type environments only in the universal system, while $\nabla, \nabla' : \mathbf{Ide} \to \mathcal{P}(\mathbf{Ide})$ is used to range over type environments only in the Amtoft-Banerjee system. Since by set isomorphism the cartesian product of two sets $\mathbf{A}, \mathbf{B}$, i.e., $\mathbf{A} \times \mathbf{B}$ approximates to the pointwise mapping $\mathbf{A} \to \mathcal{P}(\mathbf{B})$,

a set of independencies, $T^{\#} \in \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$, in the Amtoft-Banerjee framework can be represented as $\mathbf{Ide} \to \mathcal{P}(\mathbf{Ide})$. Thus an Amtoft-Banerjee system C with **Context** G can, without loss of generality, be translated into $G \vdash \nabla\{C\}\nabla'$. Using pointwise reverse inclusion, $G \vdash \nabla\{C\}\nabla'$ holds iff $\forall x \in \mathbf{Ide} \cdot \nabla(x) \supseteq \nabla'(x)$. Since both the set of independence relations and the set of dependence relations between variables in **Ide** partition the variable space, clearly, for each Δ, there is a corresponding ∇ such that $\nabla(x)$ is the complement of $\Delta(x)$ and vice versa. Thus, [75] concluded that their universal flow type system is a DeMorgan dual of the Amtoft-Banerjee system. This we graphically illustrate in Figure 2.10. The flow-sensitive type system

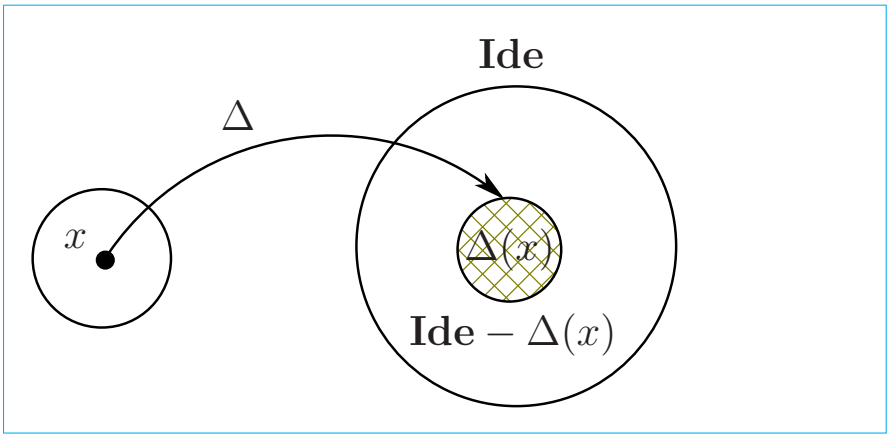


Figure 2.10: De Morgan Dual: Dependence and Independence Analysis

introduced by Hunt and Sands has all the advantages of the Amtoft-Banerjee framework. In addition, because it is based on dependence relations between variables rather than independence (as in the Amtoft-Banerjee framework), it admits a simpler and more straightforward correctness proof than the latter, wherein non standard trace semantics was employed [75]. However, like many other type systems in the literature, [75] has the limitation of only dealing with a simple imperative '*While*' language, with no consideration for nondeterministic commands, for example. In fact, a direction for further development identified by Hunt and Sands [75] in conclusion of their work is to extend the framework to cover a richer language with methods, pointers, interprocedural variable independencies, et cetera. Hence, the system is limited with respect to application to real world specification / programming languages.

We conclude our discussion of security properties in the literature with a summary of the limitations of existing possibilistic security properties in Section 2.2.3 below.

2.2.3 Limitations of Existing Confidentiality Properties

In this section we summarise the limitations of existing confidentiality properties, some of which we have discussed under the respective properties. We also show the motivation for our approach to solving the problem of the preservation of security properties from specification through to implementation. These limitations include:

- *Property not preserved through step-wise refinement to implementation*. Some of the security properties that suffer from this limitation are GNI, WNI, Restrictiveness, ND.
- *Property not compositional*. This limitation affects NI, GNI, WNI, P_BNDC, S_BNDC.
- *Property applies only to deterministic non-interruptible systems*. The security properties that manifest this limitation include NI and Read-Write frame based NI.
- *Property offers no guarantee of absence of information flow*. Examples of properties with this limitation include Separability and Restrictivenes.
- *Property based on impractical assumptions (e.g., no feedbacks), hence has limited usability*. Examples include NI, GNI.
- *Property provides no clear indication of how information flow is to be prevented*. This weakness is found in the formulation of such properties as NI, Nondeducibility, Separability.
- *Property permits intuitively insecure systems*. Example is ND
- *Property unnecessarily restrictive* (e.g., by preventing flow from low security objects to high security objects), hence less usable. Examples are SNI, ND, RES, SEP.

Clearly, many existing security properties are plagued with many limitations. In addition, the abundance of different formalisms, often using different notations and frameworks makes it even more difficult to lift these theoretical work into the realms of implementation in industry. Although the authors in [97], [84] and [6] introduced general frameworks that attempt to capture all possibilistic security properties in one framework, these works for the most part offer theoretical comparisons through their frameworks, rather than a clear direction on a *practical* application in industry. Consequently, we aim to bridge this gap by using a language-based framework that is simple enough to be readily employed in industry, yet powerful enough to characterise the intuitive 'semantics' of confidentiality properties, without the luggage of limitations hanging around existing frameworks. The intuitive notions of security we desire are summed up in the following statements:

- A high security object must not write to any object that could be read by a low security object;
- The activity, or lack thereof, of a high security object must not cause any effect on low security objects' possible future views of the system.
- A low security object must not be able to read any object to which a high security object writes.
- A low security object must not be able to learn anything about high security activity through covert means such as termination behaviour, timing, resource usage, etc.

Before proceeding to present the solution we propose, however, we will in the following section discuss the meaning of programs, specifications and refinements that we adopt in this work. We also discuss therein the infamous *Refinement Paradox*.

2.3 Programs, Specifications and Refinements

The terms '*programs*', '*specifications*', and '*refinement*' have been used to mean different things in Computer Science. To avoid any ambiguity, we give in the following sections the meaning of these terms in the literature that we adopt.

2.3.1 Programs

As pointed out by Morgan in [107], the term '*program*' traditionally denotes a collection of instructions (code) to a computer, written in some precisely defined syntax (form) and semantics (meaning) constituting the language, e.g., Java, C^{++}, Haskell, etc. In this view, programs are written for the computer to interpret and use; hence they could be difficult for the human mind to grasp. In this meaning of program, the program text itself (e.g. the java statement: '*System.out.print("Hello World!");*' does not qualify as a program; it becomes a program when it executes such that it transforms the system from one state to another. This notion of *program* could be depicted as shown in Figure 2.11.

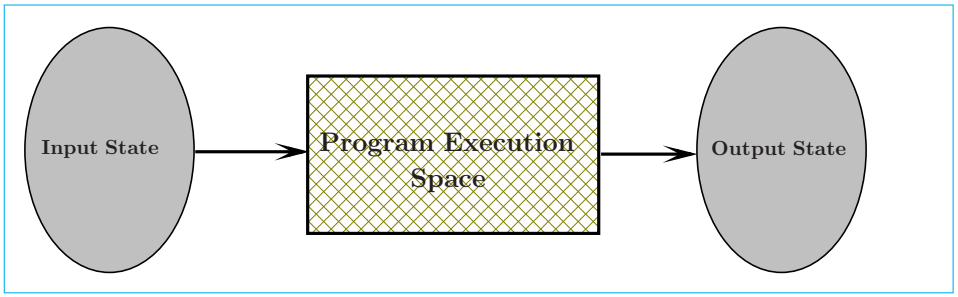


Figure 2.11: Semantics of a program (Imperative view)

Intuitively, in the view depicted in Figure 2.11, a program can be described as a contract between the programmer and the computer, which requires the computer to move the system from one state to another, provided the programmer provides the computer with the necessary environment and preconditions. However, a program could also be viewed as a specification of the requirements of the said contract between the computer on the one hand and the programmer on the other, or between the programmer on the one hand and the client on the other. In this meaning a program is *concrete*, where it formulates a contract between the programmer and the computer, and *abstract*, where it formulates a contract between the client and the programmer. We expand further on this in the following section.

2.3.2 Specifications

In its simplest form, a specification could be viewed as a contract between a client and a programmer stipulating what is required of the programmer to fulfil the contract, as well as the client's obligations. For example, the client guarantees that the user inputs satisfy the preconditions of the specification (e.g. ensuring that inputs are non-negative) whereas the programmer ensures that the postconditions are satisfied on output, given the correct inputs (e.g., the outputs are the square roots of the inputs) [31]. The client may even be the system analyst, designer, or another programmer.

The view taken by Morris in [110] is that a specification is a program, which may or may not be implementable, but have "constructs and notions that admit ease of expression". In this viewpoint, when a specification is not implementable, it is regarded as an *abstract* program, whereas when it is implementable, it is regarded as a *concrete* program, or simply, program [107]. An abstract program merely stipulates *what* the program is to do, whereas a concrete program specifies *how* a program is to do what is expected of it. Generally, a system specification begins as an abstract statement of the terms of the said "contract", which is then refined in successive steps until a more efficient and concrete program is developed. This process is termed the "*refinement process*" or "*step-wise refinement*".

Another view of specification presented, this time, by Hehner in [72] is that a specification is the expression of computer behaviour as a Boolean expression whose variables represent quantities of interest such as initial state and final (or next) state. These boolean expressions are then expected to satisfy required properties of interest. (It is worth noting that formalising computer behaviour as a Boolean expression with respect to properties of interest makes it possible to formally derive proof obligations on correctness of the program). In this viewpoint, a specification is said to be *satisfiable* or *feasible* (or correct) if it is possible for the Boolean expression to evaluate to *true* (i.e., there exists at least one final or next state resulting from some initial state); otherwise, the specification is *unsatisfiable* or *infeasible*. Generally, a specification is said to be *deterministic*, if there is only one possible solution satisfying it. Hence the solution of a deterministic specification is a function of the initial state from which it is derived. On the other hand,

where a specification is satisfied by more than one possible solution, it is considered *nondeterministic*. In this case, the solution is not a function of the input state. For example, the toss of a *fair* coin may yield 'head' on one occasion and 'tail' on another; so the result (head or tail) is not a function of the toss of the coin since a toss of a fair coin maps to two possible values.

In this thesis we adopt Morris' [110] view that specifications are *abstract programs*, with the understanding based on Hehner's conception in [72] that where such programs are deterministic, they could be viewed as functions mapping a set of possible initial states to a set of possible final states. More details of our view of refinements are discussed in the following section.

2.3.3 Refinements

The refinement process involves the application of design decisions to an abstract program (or specification) in 'successive steps' until the abstract program is transformed into a more efficient and implementable concrete program, i.e., a program that is '*at least as good as*' the original abstract program. Thus whatever transformation is made by the concrete program can also be made by the abstract one. Given that Σ denotes the set of all states in a program, S; $\sigma \in \Sigma$ ranges over *before* states while $\sigma' \in \Sigma$ ranges over *after* states. We write $\sigma \xrightarrow{s} \sigma'$ to mean that the state of the program S is changed from σ to σ' on execution of S. We define the behaviour of S, denoted $[\![\mathbf{S}]\!]$, as a set of before/after pairs that capture all possible transitions of S from all possible start states to some final states, i.e.:

$$[\![\mathbf{S}]\!] \triangleq \{(\sigma, \sigma') \mid (\sigma \in \Sigma \wedge \sigma' \in \Sigma)\ \sigma \xrightarrow{s} \sigma'\}$$

Writing $(\sigma_1, \sigma_1') \equiv (\sigma_2, \sigma_2')$ to mean that $\sigma_1 = \sigma_2 \ \wedge \ \sigma_1' = \sigma_2'$, we write Q to denote the property that:

$$\forall (\sigma_2, \sigma_2') \in [\![S_2]\!],\ \exists (\sigma_1, \sigma_1') \in [\![S_1]\!] \ \cdot \ (\sigma_1, \sigma_1') \equiv (\sigma_2, \sigma_2').$$

And let $\sqsubseteq$ denote 'is refined by', meaning in $S_1 \sqsubseteq S_2$ that program S_1 is refined by program S_2. Note that Q basically stipulates that *any behaviour manifest by S_2 must be a possible behaviour of S_1*. We write $P \vdash Q$ to denote that 'the property Q follows logically from the property P' [92], [68]. Given that $pre[S_1]$ means the precondition that must be satisfied for S_1 to be executed; $pre[S_2]$ for S_2. With this notation, the refinement relation can

be formalised as shown in Formula 2.23 below.

$$S_1 \sqsubseteq S_2 \;\Leftrightarrow\; pre[S_2] \;\vdash\; Q \tag{2.23}$$

An alternative formal representation of the refinement relation is given in Formula 2.24 below.

$$S_1 \sqsubseteq S_2 \;\triangleq\; [\![S_2]\!] \;\subseteq\; [\![S_1]\!] \tag{2.24}$$

Notice that the notion of refinement presented in Formula 2.23 means "*for S_2 to be a refinement of S_1, all states reachable by S_2 must also be reachable by S_1, and all behaviours displayed by S_2 must also be possible behaviours of S_1*", i.e., the property Q holds. Hence the set of behaviours of S_2 can be formulated as a subset of the set of behaviours of S_1 with respect to the property of interest as shown in Formula 2.24. From the foregoing we can see that the traditional notion of refinement entails preservation of total correctness[18] [54].

We say a refinement relation $\mathcal{R}$ is *defined* for programs S_1 and S_2 if for all behaviours $(\sigma_2, \sigma_2') \in [\![S_2]\!]$, there exists some $(\sigma_1, \sigma_1') \in [\![S_1]\!]$ such that $\sigma_1, \sigma_1' \equiv \sigma_2, \sigma_2'$. Thus we write $S_1 \mathcal{R} S_2$, or $(S_1, S_2) \in \mathcal{R}$ to mean that S_1 and S_2 are related by the *refinement relation* $\mathcal{R}$ whereby S_1 is refined by S_2. Another way of expressing the same idea is that S_2 *simulates* S_1. Writing $S_1 \cdot \sigma_1$ to denote that S_1 is in state σ_1, (and similar notations for S_2's states) we illustrate the refinement semantics in Figure 2.12.

A specification is expected to be logically consistent, e.g., where the specification consists of predicates over some universe (or set of states), it is impossible to derive both some proposition and its negation (i.e., a *contradiction*) from the union of all its refinements. With respect to many security properties, however, this is not always the case as we will show later on in Section 2.3.5.

[18] Total Correctness - two programs are considered equivalent under total correctness if both are guaranteed to terminate, producing the same sets of results, when executed from the same set of initial states.

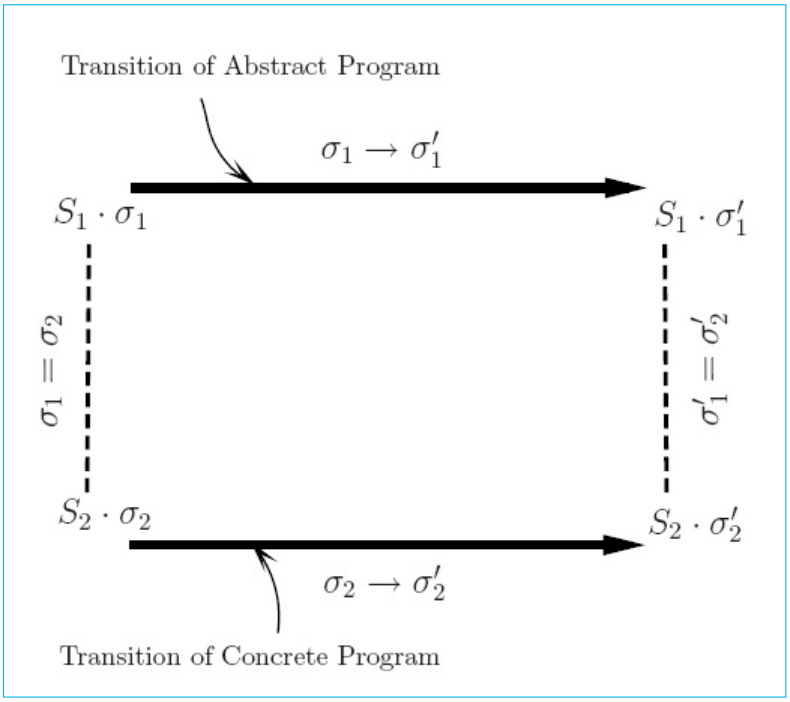


Figure 2.12: Semantics of Refinement $S_1 \mathcal{R} S_2$

2.3.4 Weakest Precondition Logic (wp)

The weakest precondition (wp) predicate transformer introduced by Dijkstra [53] in the mid-1970s is imbued with the semantics of total correctness [54], which formalises only outcomes that a program guarantees would be realised, or would not *abort*. We use the notation $wp(S_1, Q)$[19] to denote the predicate that defines the largest set of initial states from which the program S_1 is guaranteed to establish the postcondition Q. An *abort* substitution - a substitution that terminates abnormally and usually suddenly without finishing naturally, written $wp(abort, Q) = False$, is a substitution in a state outside the weakest precondition for the program of interest. Formally, a program S_2 is a refinement of another program S_1 if and only if the postcondition Q is guaranteed to be established under the given precondition. wp logic on refinement can be formalised as shown in Formula 2.25.

$$S_1 \sqsubseteq S_2 \triangleq \forall Q \cdot wp(S_1, Q) \Rightarrow wp(S_2, Q) \tag{2.25}$$

[19] An alternative notation for the weakest precondition for a program S_1 to establish the postcondition Q is written as $[S_1]Q$.

We illustrate *wp* logic on a typical substitution S in figure 2.13, where σ range over program states. Notice that σ_1 does not satisfy $wp(S, Q)$. Notice

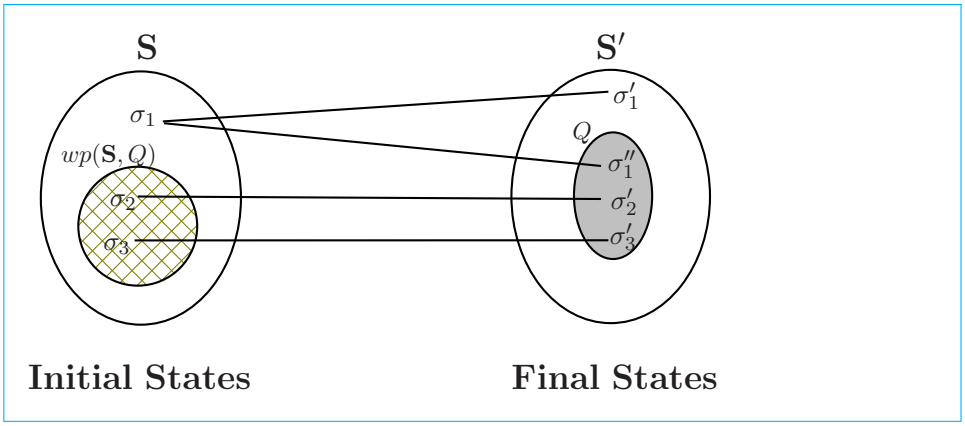


Figure 2.13: Graphical Illustration of *wp* Semantics

that there may be other valid preconditions outside the weakest precondition that satisfy program S, but do not yield the postcondition Q, e.g. σ_1 in Figure 2.14. A call, for example, to operation **a***bort* after S has started to execute may or may not yield the postcondition Q although the precondition for S may be satisfied. Thus, to avoid such unpredictability, *wp* semantics require that all programs terminate, and so it corresponds to total correctness [113], [20].

However, because wp can only be used to transform predicates for terminating programs, Dijkstra introduced another predicate transformer, *weakest liberal precondition* (*wlp*), that can deal with partial correctness semantics, although this never caught on like *wp*. Whereas *wp* requires that all programs terminate and establish the postcondition, *wlp* only requires that all terminating programs establish the postcondition, thereby allowing the possibility of some programs not terminating [53], [54], [107]. Hence, under *wlp*, a program S_2 is a refinement of another program S_1 if and only if the postcondition Q is established under the given precondition whenever S_2 terminates, i.e.,

$$S_1 \sqsubseteq S_2 \triangleq \forall Q \cdot wlp(S_1, Q) \Rightarrow wlp(S_2, Q) \tag{2.26}$$

We illustrate *wlp* logic on a typical substitution S in Figure 2.14, where we use T_s to denote *Termination set*, and $Q \wedge T_s$ captures the set of *terminating* programs that establish the postcondition Q. Notice here that σ_1 satisfies $wlp(S, Q)$ since whenever it terminates, it satisfies Q.

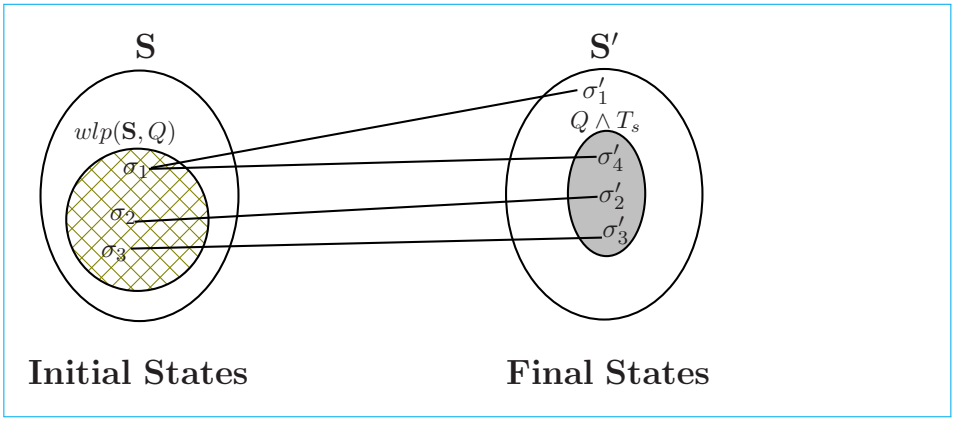


Figure 2.14: Graphical Illustration of *wlp* Semantics

Dunne et al in [54] combined *wp* and *wlp* semantics to produce what they termed *General Correctness semantics*. For consistency, we use the conventional notations of *wp* and *wlp*, employed earlier in this section, to represent the notion of General Correctness, where $GC(S_1)$ denotes general correctness with respect to S_1:

$$GC(S_1) \triangleq wp(S_1, true) \wedge wlp(S_1, Q) \tag{2.27}$$

Having discussed the notion of refinement in terms of set containment, relational semantics and weakest (liberal) preconditions, we now turn our attention to the problem of confidentiality refinement paradox in Section 2.3.5.

2.3.5 Semantics of Classical Refinement

We mentioned in Section 2.3.3 that *the refinement process* is concerned with improving on an abstract program, in successive steps until an implementable one is realised. Generally, refinement could be either *algorithmic refinement* in which case a state transformation may not be involved; or *data*

refinement where state variables are transformed [123]. Whichever type of refinement is considered, though, a refinement in its '*classical semantics*[20]' is considered valid if either of two actions (or, trivially, no action as in $S_1 \sqsubseteq S_1$) is performed on an abstract program, namely:

- Reduction or removal of nondeterminism, for example, a nondeterministic program $P > Q$ could be refined by the (less) nondeterministic program '**CHOICE** $P := Q + 1$ **OR** $P := Q + 2$ **END**' or with the deterministic program **BEGIN** $P := Q + 1$ **END**;
- Weakening of precondition, for example by reducing the assumptions and / or constraints necessary for the program to run. It should be noted that weakening the precondition in a specification favours the client, as he gets more usability from the system, whereas weakening the postcondition makes the programmer's job lighter. For example, given that **PRE** stands for 'precondition', the precondition **PRE** $n > 10$ **END** could be weakened to **PRE** $n \neq 0$ **END**.

In this thesis, we will be considering refinement as a reduction of nondeterminism (or *Underspecification* in B terminology). Thus in Subsection 2.3.5.1, we discuss in more detail the notion of refinement as a reduction of Nondeterminism.

2.3.5.1 Reduction of Nondeterminism

The primary concern of an abstract program is to capture the client's requirements, usually in a framework understandable and verifiable by the client, as documented by the requirements analysis phase of the software development process. Thus, as pointed out in Section 2.3.2, this development stage is concerned with the '*what*' rather than the '*how*' of the program. Consequently, design and implementation issues are not dealt with, but rather what the program is required to do is formalised at this stage. It makes good sense therefore to allow a measure of non-determinism by formalising only the potential behaviours of the system (*underspecification*) at this stage, leaving design and implementation concerns until later. This allows the implementer the freedom to choose among multiple possible im-

[20] Classical refinement semantics - conservative extension of refinement.

plementations.

Since an abstract program is not implementable, and on its own cannot fulfil the client's requirements, to fulfil the client's requirements the developer has to refine the abstract program systematically until an '*at least as good as' and implementable* version is realised. So during this stepwise refinement stage, the question of how the program is to achieve the client's objectives; its interaction with hardware and its environment; fault-tolerance, security, etc are considered. In addition, the designer now has to make choices between possible implementations allowed by the abstract program until he gets a program that is deterministic (usually, but not always), implementable, and hopefully satisfies the *requirements*. Semantically, therefore, the concrete program can be viewed as a subset of the abstract program in the sense that whatever behaviour is allowed by the concrete program is necessarily allowed by the abstract program. Below is an example of a valid refinement by the semantics discussed here, assuming S_A denotes a Nondeterministic generalised substitution (or assignment statement) at the abstract level; S_C denotes a deterministic substitution at implementation; [] denotes 'Nondeterministic choice'; and j, k are simply variables.

$$(S_A \triangleq j := k + 3 \ [] \ j := k + 7) \sqsubseteq S_C \triangleq j := k + 7 \tag{2.28}$$

Note, though, that certain interactive systems may require the preservation of Nondeterminism down to implementation, by providing multiple solutions from which one could be either chosen nondeterministically or in response to the environment. Haugen and Stølen [65] distinguished between this form of nondeterminism, which they termed *Explicit Nondeterminism* because it captures mandatory behaviour, and Underspecification, which only formalises potential behaviour. On explicit Nondeterminism, the authors gave the example of a lottery system where each lottery ticket has the possibility of winning some prize, hence every behaviour manifested by the different tickets must be implemented for the system to be fair.

2.3.5.2 Confidentiality Refinement Paradox

In the process of refinement it is important that properties of interest such as correctness, safety, etc, be preserved by the new and "better" versions of

the system as one moves down the refinement tree towards an implementable program. With security properties, though, it is well known that they are generally not preserved through refinement. One key reason for this is that when an abstract specification allows nondeterminism by underspecification it may satisfy certain security properties like GNI, WNI, etc, but when the potential behaviours are refined away to get a deterministic implementation, the implementation tends to fail the corresponding deterministic security property such as NI. This situation whereby a system specification that on implementation loses a security property that it satisfies at the abstract level is termed *confidentiality refinement paradox*. We highlight this problem by a simple example, namely:

> Consider an abstract specification intended to arbitrarily select a number from the set of natural numbers, and assign this to a variable h, and we desire the number thus assigned to h to remain a secret.

Given that n ranges over the set of natural numbers $\mathbb{N}$, in the specification, S_A, we then formulate S_A as:

$$S_A \;\triangleq\; \textbf{PRE } n \in \mathbb{N} \textbf{ THEN } \;\; h := n \textbf{ END} \tag{2.29}$$

Notice in Substitution 2.29 above that it is impossible to deduce the value stored in h by the program because of the nondeterminism arising from the assignment of an arbitrary number, n, which range over a large set, in this case, the set of natural numbers. By the semantics of classical refinement, however, since $7 \in \mathbb{N}$, a valid refinement of Substitution 2.29 is:

$$S_C \;\triangleq\; \textbf{BEGIN } h := 7 \textbf{ END} \tag{2.30}$$

Clearly,

$$S_A \sqsubseteq S_C, \text{ since } [\![(h := 7)]\!] \subseteq [\![(\textbf{PRE } n \in \mathbb{N} \textbf{ THEN } \;\; h := n \textbf{ END})]\!] \tag{2.31}$$

Notice that unlike the abstract specification in Substitution 2.29, the concrete refinement given in Substitution 2.30 is no longer secure, because the value assigned to h can be learned by an adversary that has access to the program text. This is one example illustrating that confidentiality proper-

ties is generally not preserved through refinement.

We will in Section 2.4 present an example using a formal specification language, The B Method, to show that even the precision of formalism does not in itself guarantee the preservation of confidentiality properties. In the following section, however, we present a summary of the limitations of the traditional refinement semantics.

2.3.5.3 Limitations of the Classical Refinement Relation Semantics

We summarise below the limitations, with respect to security properties, of the classical refinement relation:

- The classical refinement relation was formalised mainly with the preservation of functional program properties based on linear temporal logic in mind, with little consideration for security properties, which are properties of multiple runs or traces of the system [6], [97];
- It is virtually impossible to capture security properties and preserve them through refinement to implementation using the classical refinement relation alone;
- Classical refinement relation assumes total correctness thereby factoring out the possibility of non-termination. Thus one cannot prevent covert flows due to termination behaviour while using the classical refinement relation alone;
- It is difficult to formalise dynamic/interactive behaviour using traditional refinement mechanisms such that changes in security classification of variables could be correctly propagated from specification through to implementation, because interactivity may introduce additional visibility between variables that may leak secure information while not breaching the classical refinement relation.

In view of these limitations, clearly there is a need to '*do something*' to empower developers to be able to securely refine software systems. A few researchers like Mantel [99], Alur et al [6], Jürjens [82], Seehusen and Stølen [129], and Carroll Morgan [108], [109] have proposed some extensions to the

classical refinement relation. These proposals are discussed in the following section.

2.3.5.4 Existing Confidentiality-Preserving Refinement Frameworks

Mantel's Refinement Operators for PSP.
Mantel in [99] introduced a couple of refinement operators for Perfect Security Property, PSP. This is one of the major works that we know on Confidentiality-preserving refinement frameworks. Recall that the author gave two unwinding conditions (see Formulae 2.6 and 2.7 in Section 2.2.2.8), satisfaction of which is sufficient proof that the refinement also satisfies PSP. Thus the aim of the refinement operators introduced in [99] is to constrain state-event systems such that only refinements that satisfy these unwinding conditions are considered to be valid. The two refinement operators introduced in this work [99] are $\overline{refine}$ and $\underline{refine}$. Given that Σ, E, SES denotes set of *states*, set of *events*, and state-event systems respectively, each of these refinement operators take three parameters, namely:

- The system to be refined formalised as a SES;
- A set of state-event pairs, $DS \subseteq \Sigma \times E$, some of which would be disabled during refinement; and
- The unwinding relation $\approx_L \subseteq \Sigma \times \Sigma$

While $\overline{refine}$ disables some, possibly not all, of the state-event pairs in DS, $\underline{refine}$ disables all state-event pairs in DS, possibly along with some pairs external to DS.

To disable either high-level or low-level state-event pairs, the functions $Hdisable$, $\underline{Ldisable}$, and $\overline{Ldisable}$ are used in constructing the refinement relations introduced by the author. Given that $T_R \subseteq \mathbf{\Sigma} \times E \times \mathbf{\Sigma}$ denotes the set of transitions of a state event system from one state to another as a result of the execution of an event or events; $DS_H \subseteq \Sigma \times H$ denotes the set of high level state-event pairs ($\Sigma \times H$); $DS_L \subseteq \Sigma \times L$ denotes the set of low level state-event pairs ($\Sigma \times L$); the new transition relation on the state-event systems, denoted $\underline{disable}$ and $\overline{disable}$ are functions constructed from three parameters T_R, DS, and $\approx_L$. Writing $Hdisable(T_R, DS_H)$ the author defines the set

of transitions T_R with DS_H disabled. Mantel's notion of secure refinement relation is built on the functions, $Hdisable$, $\underline{Ldisable}$, $\overline{Ldisable}$, $\underline{disable}$ and $\overline{disable}$, formally defined as follows:

Function 1 ($Hdisable$)**.** *disables all high level state-event pairs in all system transitions, as shown formally below*

$$Hdisable(T_R, DS_H) \;=\; \{(\sigma_1, e, \sigma_2) \in T_R \mid (\sigma_1, e) \notin DS_H\}$$

Function 2 ($\underline{Ldisable}$)**.** *For all states $\{\sigma' | (\sigma', l) \in DS_L\}$, disabling l in σ' requires that l be disabled also in all low-equivalent states, i.e., all states $\{\sigma \mid \sigma \approx_L \sigma'\}$. This notion captured by $\underline{Ldisable}$ is depicted formally below.*

$$\begin{aligned}\underline{Ldisable}(T_R, DS_L, \approx_L) \;=\; \{&(\sigma_1, e, \sigma_2) \in T_R \mid (\sigma_1, e) \notin DS_L \;\;\wedge \\ &\neg\exists (\sigma_1', \sigma_2') \in S \cdot \\ &(\sigma_1 \approx_L \sigma_1' \;\wedge\; (\sigma_1', e, \sigma_2') \in T_R \;\wedge\; (\sigma_1', e) \in DS_L)\}\end{aligned}$$

Function 3 ($\overline{Ldisable}$)**.** *For all states $\sigma' \mid \sigma' \approx_L \sigma$, a low level event l may only be disabled if $(\sigma, l) \in DS_L$ and $(\sigma', l) \in DS_L$. The equation below formally captures this notion.*

$$\begin{aligned}\overline{Ldisable}(T_R, DS_L, \approx_L) \;=\; \{&(\sigma_1', e, \sigma_2') \in T_R \mid (\sigma_1', e) \in DS_L \;\Rightarrow \\ &\exists \sigma_1, \sigma_2 \in S \cdot \\ &(\sigma_1 \approx_L \sigma_1' \;\wedge\; (\sigma_1, e, \sigma_2) \in T_R \;\wedge\; (\sigma_1, e) \notin DS_L)\}\end{aligned}$$

Function 4 ($\underline{disable}$)**.** *This function is basically a conjunction of both $Hdisable$ and $\underline{Ldisable}$ as shown in the following formula.*

$$\underline{disable}(T_R, DS, \approx_L) \;=\; Hdisable(T_R, DS_H) \;\cap\; \underline{Ldisable}(T_R, DS_L, \approx_L)$$

Function 5 ($\overline{disable}$)**.** *Similarly, this function is a conjunction of both $Hdisable$ and $\overline{Ldisable}$ as shown in the equation below*

$$\overline{disable}(T_R, DS, \approx_L) \;=\; Hdisable(T_R, DS_H) \;\cap\; \overline{Ldisable}(T_R, DS_L, \approx_L)$$

Using the functions given above, [99] proposed the two refinement relations given in equations(2.32 and 2.33) below.

$$\underline{refine}(SES, DS, \approx_L) \;=\; (\Sigma, \Sigma_I, E, I, O, \underline{disable}(T_R, DS, \approx_L)) \qquad (2.32)$$

$$\overline{refine}(SES, DS, \approx_L) \;=\; (\Sigma, \Sigma_I, E, I, O, \overline{disable}(T_R, DS, \approx_L)) \qquad (2.33)$$

Notice the correlation between these definitions and the configuration given in Section 2.2.2.8.

Jürjens' Secrecy-Preserving Refinement

Jan Jürjens in [82] presents work towards a framework for stepwise development of secure systems using a notion of secrecy to be preserved by traditional refinement operators. The author pointed out that because many proposed security properties in the literature are properties of *sets of traces*, rather than properties of traces, these properties are often not preserved under refinement. Subsequently, [82] argued that developing secure systems for such properties in a stepwise manner requires to redo security proofs at each refinement step. Thus, verifying such properties at specification level may not necessarily guarantee that the property will be satisfied at the implementation level.

The author made a distinction between the two types of *Nondeterminism*, namely: *Underspecification*, whereby details may be left out in early phases of system development, and *Unpredictability*, which is a vital part of the functionality of a system, and is used for security, e.g., selection of keys or passwords. Since providing security through underspecification is akin to providing *security by obscurity*, [82], nondeterministic choice of functional importance is recommended for specifications that require security-preserving refinement, rather than the conventional nondeterministic choice operator, which does not distinguish between underspecification and unpredictability. A similar point was made by Morgan in [108], [109].

The secrecy property considered in [82] relies on the notion that a process specification preserves the secrecy of a piece of data d if the process never sends out any information from which d could be derived, even in interaction with an adversary. As acknowledged by the author, this is a less fine-grained notion of secrecy property as those based on equivalence relations, and hence may not prevent *implicit flows*.

The author modeled secrecy on communicating processes interacting by transferring sequences of data values over unidirectional FIFO communicating channels. Processes are viewed as collections of programs that communicate synchronously (in rounds) through channels, with the constraint that for each of its output channels c the process contains exactly one program p_c that outputs on c. The intuition captured is a description of a value to be output on the channel c in the $n+1^{th}$ round, computed from values on channels in the n^{th} round [82]. Given that $I \subseteq \mathbf{Channels}$ is a set of input channels, $O \subseteq \mathbf{Channels}$ is a set of output channels, $L \subseteq \mathbf{Channels}$ is a set of local channels used to store local state between execution rounds, and $c \in O \cup L$ is an output on channel c, a process is defined as a tuple $P = (I, O, L, (p_c)_{c \in O \cup L})$.

A *stream processing function* $f : \mathbf{Stream_I} \to \mathcal{P}(\mathbf{Stream_O})$ is defined as a mapping from streams to sets of streams. And given that $i = 1, 2$, $O_1 \cap O_2 = \varnothing$, $I = (I_1 \cup I_2) \backslash (O_1 \cup O_2)$, and $O = (O_1 \cup O_2) \backslash (I_1 \cup I_2)$, the composition of two stream-processing functions $f_i : \mathbf{Stream_{I_i}} \to \mathcal{P}(\mathbf{Stream_{O_i}})$ is defined as

$$f_1 \otimes f_2 : \mathbf{Stream_I} \to \mathcal{P}(\mathbf{Stream_O})$$

Since the binary composition operator $\otimes$ is both associative and commutative, [82] defined a generalised composition operator $\bigotimes_{i \in I} f_i$ over a set $\{f_i : i \in I\}$ of stream-processing functions. A process $P = (I, O, L, (p_c)_{c \in O})$ is then modeled by a stream-processing function $[\![P]\!] : \mathbf{Stream_I} \to \mathcal{P}(\mathbf{Stream_O})$ from input streams to sets of output streams. The author's notion of secrecy is then defined in terms of stream-processing functions as follows.

Given that I_P, O_P are sets of input and output channels respectively of the process P; I_A, O_A are similarly sets of input and output channels of process A. $S_A \subseteq \mathbf{Secret}, K_A \subseteq \mathbf{Keys}$ are, with respect to process A, sets of secrets and keys respectively, and $[\![A]\!]_r$ denotes the composite stream function of

A. A process P is adjudged to leak a secret $m \in \mathbf{Secret} \cup \mathbf{Keys}$ if there is a process A with $I_A \subseteq O_P,\ I_P \subseteq O_A$ and $m \notin S_A \cup K_A$ such that $[\![P]\!] \otimes [\![A]\!]_r$ may eventually output m. Otherwise, P preserves the secrecy of m. Notice that the input to the adversarial process A, I_A, is a subset of the output of P, and the output of A, O_A, is fed into the input of P, I_P. That is, P preserves the secrecy of m if no adversary can find out m in interaction with P. The author then defined their refinement property, where $\rightsquigarrow$ denotes "*is refined by*", as follows.

> For processes P and P' with $I_p = I_{p'}$ and $O_p = O_{p'}$,
> $P \rightsquigarrow P'$ if for each $s \in \mathbf{Stream_{I_P}},\ [\![P]\!](s) \supseteq [\![P']\!](s)$.

This refinement property is adjudged secure under the following conditions

- P' preserves the secrecy of m iff P preserves the secrecy of m and $P \rightsquigarrow P'$.
- For any $C \subseteq \mathbf{Stream_{O_P}} \times \mathbf{Stream_{I_P}}$, P' preserves the secrecy of m assuming C iff P preserves the secrecy of m assuming C.

Inferable Properties Based Secrecy Refinement (IBS)

Alur et al in [6] proposed a formal and general simulation-based proof technique for refinement that also takes into account the notion of secrecy as one of the properties of interest to be preserved in a software system. The attacker model employed is that the observer knows the specification of the system but can only see a subset of the traces of the system from which he may be able to make inferences about secrecy properties. Using a standard labeled transition system, the authors formalised their notion of secrecy based on three parameters, namely: an equivalence relation (used to capture what observers could discern about the system) on runs of the system; the properties to be kept secret; and the set of all runs of interest.

Intuitively, an implementation is a secure refinement of a specification if for every run r of the implementation, there is a run r' of the specification such that the observer cannot distinguish r' from r, and for every property that the observer can deduce from r in the implementation, it is the case that such properties are also deducible from the run r' of the specification. The authors strengthened the refinement notion of trace inclusion, whereby a

program P is said to be refined by a program Q if and only if $[\![Q]\!] \subseteq [\![P]\!]$, or P *simulates* Q. They in effect discard some behaviours that satisfy the classical refinement relation but does not satisfy their framework. Their secure refinement framework is akin to the notion of bisimulation in the literature because it is *reflective*. That is, for Q to be a *secure* refinement of P, it must be the case that both programs are *bisimilar* in the sense that: for Q to refine P, P must *simulate* [21] Q; and for the refinement to be secure, the simulation must be reversible, i.e., Q, (the refinement) must also simulate P (the original specification). It should be noted, though, that this backward simulation is not in itself *sufficient* to guarantee secure refinement. The authors suggest instead that the behaviours of the implementation must be a subset of the behaviours of the original specification, while at the same time the specification must simulate the implementation.

Given that Q denotes a set of states; $I \subseteq Q$ denotes a set of initial states; L a set of labels; and $\delta \subseteq Q \times L \times Q$ is a transition relation on the set of states. A labeled transition system (LTS), T, is defined as a tuple (Q, L, δ, I). A run $r = q_0 l_0 q_1$ is a sequence of alternating states and labels where $q_0 \in I$ and $\forall i \quad 0 \leq i < |r| \Rightarrow (q_i, l_i, q_{i+1}) \in \delta$, and $R(T)$ denotes the set of all runs of the LTS T. A property $\alpha \subseteq R(T)$ is a state property if and only if it, i.e. α, depends *only* on the last state of a run. Using a tripartite domain $\{\top, \bot, m\}$ corresponding to *True, False, maybe* respectively, the property models the observer's knowledge of the system as an equivalence relation on the set of runs, and the property α is not secret if the observer can conclude that α holds or not. However, whenever the observer is unable to conclude whether α holds or not, then the the property is considered secret. Thus using the equivalence relation $\equiv \subseteq R(T) \times R(T)$ to denote the observer's knowledge of the property α after run r, the authors defined a function termed *Inferable Properties*, IP, as follows:

$$IP(r, \alpha, \equiv) = \top \;\Leftrightarrow\; \forall r' : r' \equiv r \Rightarrow r' \in \alpha$$

$$IP(r, \alpha, \equiv) = \bot \;\Leftrightarrow\; \forall r' : r' \equiv r \Rightarrow r' \notin \alpha$$

[21] P simulates Q means that whatever behaviour exhibited by Q must be a possible behaviour of P, i.e., it must be possible for P to 'do anything that Q can do'.

Otherwise, $IP(r, \alpha, \equiv) = m$

The first two cases indicate the observer is able to learn something about α after the run r, whereas the third case shows the observer is unable to learn anything about the property α after the run r, hence it is only in this instance that the system can be adjudged secure with respect to the property of interest. To apply this framework to Noninterference, the authors first formalised the notion of Noninterference as a functional equivalence $\approx_f$ on system inputs and outputs. This derives from the intuition that Noninterference implies that 'the execution of a program from two states that share the same input values of low security variables manifest behaviours indistinguishable from the observer's view. PSP is similarly modeled as an equivalence relation $\approx_{psp}$ on the set of possible traces observable to the low security observer. Given that $\sqsubseteq$ denotes 'is refined by', $\leq$ is a partial ordering on the observer's knowledge, IP, of some properties of interest, we can say of two programs T_{spec} and T_{imp} that: $T_{spec} \sqsubseteq T_{imp}$ if for each run r of T_{imp}, there is an equivalent run r' of T_{spec} such that the observer can deduce less about the properties of interest from observing T_{imp} execute r than from observing T_{spec} execute r'.

Formally, $\quad T_{spec} \sqsubseteq T_{imp} \;\Leftrightarrow\; IP(r, \alpha, \equiv) \leq IP(r', \alpha, \equiv).$

With some interesting examples, [6] showed how some existing security properties can be formalised in their framework and argued that their framework is general enough to capture all possibilistic security properties in the literature. However, they noted that their work fall short of providing designers with a tool that can help them transform programs in such a way that secrecy properties are preserved through refinement. This is one area we address in our framework presented in Section 3.4. For now, we review the confidentiality-preserving refinement framework introduced by Banks and Jacob [19], modellled using Hoare and He's Unified Theories of Programming (UTP) .

Confidentiality-preserving Refinement for UTP.
Banks and Jacob in [19] introduced a novel way of encoding confidentiality properties using Hoare and He's Unified Theories of Programming (UTP).

They devised conditions for verifying that system designs do not leak secret information to untrusted users. This work also proposed how the derived confidentiality conditions can be combined with the traditional notions of refinement to yield refinement relations suitable for ensuring that systems are securely implemented. Unifying Theories of Programming deals with program semantics. It shows how denotational semantics, operational semantics and algebraic semantics can be combined in a unified framework for the formal specification, design and implementation of programs and computer systems [71], [145]. The formal representation of the UTP is a relational calculus expressed as predicates over an *alphabet* or a set of observational variables (i.e., all information about a program's execution that is visible to observers) of the UTP.

The authors described an observation of a UTP predicate S as any predicate that maps each variable in the alphabet of S to a single value, such that S is satisfied by that mapping. The authors distinguished between two classes of observation, namely: a *system-level* observation, and an *interface-level* observation. A system-level observation of a UTP predicate S describes all possible behaviour of S, whereas an interface-level observation of S, *disjoint* from a system-level observation, describes a user's observation of the system. Given that s denotes a list of variables $s_1 \ldots s_2$, s' stands for the list of variables $s'_1 \ldots s'_2$, and let Φ be a system-level observation of S. The notion that Φ is satisfied by exactly one valuation of the system-level variables, and that Φ is an actual observation of S is captured in the following predicate.

$$(\exists_1 s, s' \bullet \Phi) \ \wedge \ [\Phi \Rightarrow S]$$

A predicate that formalises a user's interface to a system by defining a total relation (or function, in deterministic settings) from system-level observations to interface-level observations is termed a view. Banks and Jacob, [19], insisted that views associated with different users share no interface-level observational variables, and that a view should not restrict the domains of the system-level observational variables. Given a system-level predicate S, and a view V, the interface-level observations that can be made by monitoring the behaviour of S through V (i.e., the image of S as projected through V), denoted $\mathbf{P}(V, S)$ is encoded in the following formula:

$$\mathbf{P}(V, S) \ \triangleq \ \exists s, s' \bullet V \wedge S$$

Given that $\mathcal{H}$ denotes a high-security user whereas $\mathcal{L}$ denotes a low-security user, S is considered secure if, for any observation of S that $\mathcal{L}$ can make, $\mathcal{L}$ is unable to deduce confidential information about $\mathcal{H}$. This requirement is parameterised on the *restriction* predicate and the *closure* requirement. The restriction predicate, R, is defined over $\mathcal{H}$'s observational variables denoted u_H, u'_H, i.e., the space of $\mathcal{H}$ observations featuring confidential activities. Thus, a $\mathcal{H}$ observation is only classed confidential if and only if it appears in R. The closure requirement, Q, is a predicate relating confidential $\mathcal{H}$ observations in R to alternative $\mathcal{H}$ observations that are not classed as confidential [19]. Whenever Q relates a confidential activity ψ to a non-confidential activity $\tilde{\psi}$ (both in $\mathcal{H}$) such that ψ and $\tilde{\psi}$ are indistinguishable to $\mathcal{L}$, the authors say $\tilde{\psi}$ is a *cover story* for ψ. It is necessarily assumed then that the domain and co-domain of the relation encoded by Q are disjoint, i.e., cover stories cannot themselves be classed as confidential. Given that H and L denote the views of $\mathcal{H}$ and $\mathcal{L}$ respectively, R is a restriction and Q a closure requirement over H, the notion of confidentiality properties, denoted π, in the paper is captured as a tuple $\pi = \langle H, L, R, Q \rangle$. Consequently, a system S is said to satisfy a confidentiality property $\pi = \langle H, L, R, Q \rangle$ if and only if the following formula holds.

$$[\mathbf{P}(L \wedge H, S) \wedge R \Rightarrow (\exists\, \tilde{u_H}, \tilde{u'_H} \bullet \mathbf{P}(L \wedge \tilde{H}, S) \wedge Q)] \tag{2.34}$$

The notion that, for every $\mathcal{H}$ activity ψ in a system T classed as confidential by R with respect to the confidentiality properties π, all the cover stories related to ψ by Q that are present in S must also be present in T is captured by writing $S \trianglelefteq_\pi T$, which is formally defined as:

$$S \trianglelefteq T \;\triangleq\; \mathbf{P}(L \wedge \tilde{H}, T) \sqsubseteq \mathbf{P}(L \wedge H, T) \wedge R \wedge \mathbf{P}(L \wedge \tilde{H}, S) \wedge Q \tag{2.35}$$

Following from that, [19] proposed a confidentiality preserving refinement, denoted $\sqsubseteq^{cp}_\pi$, with respect to the confidentiality property π as:

$$\sqsubseteq^{cp}_\pi T \;=\; S \sqsubseteq T \wedge S \trianglelefteq_\pi T \tag{2.36}$$

Maintaining Information Flow Security under Refinement

One of the key contributions of Seehusen and Stølen in [129] is the problem of maintaining information flow security during the refinement process,

which they dealt with via a schema they formulated for specifying secure information flow properties that are preserved by their notion of refinement. The authors modeled the input-output behaviour of systems by finite sequences of events called *traces*. A set of events is denoted $\mathcal{E}$. An event is viewed as either the *transmission* or *reception* of a message, and given that k, m denotes a kind and a message respectively, an event is formally described as a pair (k, m). An event kind is either a message transmission, denoted !, or a message reception, denoted ?. Given that a_1 is a transmitter, a_2 is a receiver, and s denotes a signal, a message is formalised as a triple (a_1, a_2, s). Given that $\sqsubseteq$ denotes the standard prefix ordering on sequences, [129] required that the semantics of a system specification be a prefix-closed set of traces, i.e., where t, t' are traces and A is a set of traces,

$$(t \in A \wedge t' \sqsubseteq t) \;\Rightarrow\; t' \in A$$

Given that $\mathbb{P}(A)$ denotes the powerset of A, i.e., $\mathbb{P}(A) = \{X | X \subseteq A\}$; a trace is written as $\langle e_1, e_2, \ldots, e_n \rangle$, and an empty trace is written as $\langle\rangle$. The projection of a trace t on a set of events E, denoted $t|_E$, is derived by deleting all elements not in E. The authors described information flow security (a restriction on information flow between two observers) as a *flow policy* and a *secure information flow predicate*, also termed security predicate. Where $\mathcal{O}$ is a set of observers and o_1, o_2 are individual observers, the flow policy is formally defined as a relation $\nrightarrow \subseteq \mathcal{O} \times \mathcal{O}$. Thus, the flow policy $(o_1, o_2) \in \nrightarrow$ means information shall not flow from o_1 into o_2. Security predicates are defined in terms of what observers can perceive. Hence, an observer o of a trace t may only perceive information in t with all events not in o deleted, i.e, given that $E.o$ denotes the set of all events that can be observed by o, the authors write $t|_{E.o}$. Thus, given that $\mathcal{E}$ denotes the set of all events, the events that can be transmitted (or received) by o is formally defined as:

$$E.o \;\triangleq\; \{(k, (a, a', m)) \in \mathcal{E} \mid (k =! \wedge a = o) \;\vee\; (k =? \wedge a' = o)\} \tag{2.37}$$

To prevent implicit flows, the authors employed the notion of *low-level indistinguishability*, whereby two traces t and t' are considered *indistiguishable* from a low-level user, L's point of view, written $t \sim_I t'$, if and only if the set of all events that can be observed by L ($E.L$) in t is equal to that which can be observed by L in t'. Formally, this is defined as:

$$t|_{E.L} \;=\; t'|_{E.L}.$$

This notion of indistinguishability was then used to construct a *low-level equivalent set* LLES, a notion introduced in [84]. Thus, given a specification Φ, L may, from an observation $t \in \Phi$, construct a LLES and infer that some trace in the LLES has occurred, but not which one. In this framework, the LLES is formally defined as:

$$\{t' \in \Phi \mid t \sim_I t'\}$$

Seehusen and Stølen [129] illustrated the requirement that any LLES that L can construct be *closed* with respect to some criteria with the following example:

Example: Let $\Phi = \{\langle\rangle, \langle l_1\rangle, \langle h_1\rangle, \langle h_2\rangle, \langle h_1, l_1\rangle, \langle h_1, l_2\rangle, \langle h_2, l_2\rangle\}$, and assume that L may only observe l_1 and l_2 and that h_1 and h_2 are high-security events. Given a security definition that L may not, with certainty, be able to infer from his observation of low-security events that a high-security event has occurred, an observation of l_1 is secure because L may infer that either $\langle l_1\rangle$ or $\langle h_1, l_1\rangle$ has occurred, but cannot, with certainty, infer that $\langle h_1, l_1\rangle$ has occurred. An observation of $\langle l_2\rangle$, however, is insecure because from this L can deduce that either $\langle h_1, l_2\rangle$ or $\langle h_2, l_2\rangle$ has occurred. Thus, in either case, L can infer with certainty that a high-security event has occurred.

From the example above, it is obvious that a specification $\Phi' = \{\langle h_1, l_2\rangle, \langle h_2, l_2\rangle\}$ *refines* Φ, but as discussed in the example, Φ is insecure. A secure specification is formalised based on Mantel's *Basic Security Predicates* (BSP) [101]. BSP demands that for any trace t of the specification Φ there must be another trace t' such that $t \sim_I t'$ from L's view, and which fulfills a closure condition Q, the *closure requirement* of BSP. This is formally defined as follows.

A specification Φ satisfies the basic security predicate, $BSP_{QR}(\Phi)$ for restriction R and closure Q iff

$$\forall t \in \Phi \cdot R(\Phi, t) \Rightarrow \exists t' \in \Phi \cdot t \sim_I t' \wedge Q(t, t') \tag{2.38}$$

Notice there is a touch of *unpredictability* with respect to L's inference whenever he observes l_1 in the example above. Hence the closure requirement,

Q, may be viewed as the security predicate's requirement for unpredictability, since refinements that remove traces which provide this unpredictability may make the refinement insecure (*refinement paradox*). Thus, [129] introduced a new notion of refinement whereby a distinction is made between *underspecification* and *unpredictability*. This allows underspecification with respect, for example, to data refinement to be refined away, while never completely removing unpredictability via what is often termed *property refinement* [33], [32]. This notion of unpredictability is captured by defining a *prefix-closure* on the set of traces. That is, given that Ω denotes a system specification (or a set of trace sets), where each trace set ϕ is termed an *obligation*, and $\widehat{\Omega} = \bigcup_{\Phi \in \Omega} \Omega$, it is required that for all traces t, t'

$$t \in \widehat{\Omega} \wedge t' \sqsubseteq t \;\Rightarrow\; t' \in \widehat{\Omega}$$

Hence this notion of refinement requires that a system Ω' is a refinement of another system specification Ω, written $\Omega \rightsquigarrow \Omega'$, iff

$$(\forall \phi \in \Omega \cdot \exists \phi' \in \Omega' \cdot \phi' \subseteq \phi) \wedge (\forall \phi' \in \Omega' \cdot \exists \phi \in \Omega \cdot \phi' \subseteq \phi)$$

Given a specification Ω, a restriction R, and a closure requirement Q, the authors subsequently redefined their notion of basic security property given in Formula 2.38 as follows, asserting that Ω satisfies $BSP_{QR}(\Omega)$ iff

$$\forall t \in \widehat{\Omega} \cdot R(\widehat{\Omega}, t) \;\Rightarrow\; \exists \phi \in \Omega \cdot \forall t' \in \phi \cdot t \sim_I t' \wedge Q(t, t') \tag{2.39}$$

Consequently, the notion of secure refinement introduced by [129] was formalised as

$$\Omega \rightsquigarrow \Omega' \wedge BSP_{QR}(\Omega) \;\Rightarrow\; BSP_{QR}(\Omega') \tag{2.40}$$

Carroll Morgan's Ignorance-preserving Refinement

Carroll Morgan in [108], [109] modeled the concept of secrecy as an adversary's ignorance of concealed information. The author also introduced an extension to the classical refinement relation based on identifying and reasoning about an adversary's ignorance of hidden data. Programs are assumed to have their state space partitioned into a *visible* part v and a *hidden* part h. A *weak* adversary with knowledge of program text and the ability to see initial and final values of v was initially assumed. However,

using *Gedanken experiments* [22], [109] argued that if *refinement* is allowed, one is forced to assume that the adversary can see both program flow and intermediate values of visible variables. One such experiment shows that if the program $v := h; v := 0$ is to be refined, then it is possible for the value in h to be leaked into v through a series of simplification stepwise refinements as shown below, where T denotes the set of possible values of h and v:

$$\begin{aligned} v := h; v := 0 &= (v := h; v := 0); v :\in T \\ &= v := h; (v := 0; v :\in T) \\ &= v := h; v :\in T, \text{ and} \\ v : h; v :\in T &\sqsubseteq v := h; \mathbf{skip} \\ &= v := h \end{aligned}$$

Hence, the observer can learn the value of h in the intermediate program $v := h$. Writing $\sqcap$ to denote an *atomic*[23] *nondeterministic choice* operator, Morgan shows, using the program $h := 0 \sqcap h := 1$, that an observer could *hypothetically* observe program flow, as depicted below:

$$\begin{aligned} h := 0 \sqcap h := 1 &= (h := 0 \sqcap h := 1); v :\in T \\ &= (h := 0; v :\in T) \ \sqcap \ (h := 1; v :\in T) \\ &\sqsubseteq (h := 0; v := 0) \ \sqcap \ (h := 1; v := 1) \\ &= (h := 0; v := h) \ \sqcap \ (h := 1; v := h) \\ &= (h := 0 \sqcap h := 1); v := h \end{aligned}$$

Consequently, the author concluded that a *strong* adversary must be modeled, who knows at every program counter what program steps have occurred, and what the values of the visible variables are after each program step, not just the initial and final values.

A program's global state is assumed to comprise both current visible $\overline{v}$ and current hidden $\overline{h}$ variables, and a history sequence $\overline{p}$ of the program counter. The possible runs of a system are considered to be all sequences of global states that could be produced by the successive execution of atomic steps from some initial state v_0, h_0. Conceptually, in knowledge-based reasoning,

[22] In Physics, a *Gedanken experiment* is a hypothetical ("thought") experiment which is possible in principle and is analysed (but not performed) to test some hypothesis. It is also known as thought experiment [95].

[23] An atomic program is a program that can be executed in a single *indivisible* step, and changes the program counter when executed.

an observer knows all possible runs of a system, but is ignorant of how runs are interleaved.

Hence, if an observer knows a fact in a run of the system starting from a given global state, it must be the case that he knows that fact in all other possible runs of the system from the start state. Thus, if the current state of a system S is $(\overline{v}, \overline{h}, \overline{p})$, then the set of all possible states associated with the current state is the set of all other triples $(\overline{v}, \overline{h_1}, \overline{p})$ that S could also have produced from v_0, h_0. [109] writes this equivalence relation as $(\overline{v}, \overline{h}, \overline{p}) \sim (\overline{v}, \overline{h_1}, \overline{p})$, and introduced a set-valued *shadow* variable H to hold all possible values of h in all those other runs the adversary considers $\sim$-equivalent to $(\overline{v}, \overline{h}, \overline{p})$. Ergo the abstraction to (v, h, H) is represented as:

$$v = \mathbf{last}.\overline{v} \ \wedge \ h = \mathbf{last}.\overline{h} \ \wedge \ H = \{\overline{h}' | (\overline{v}, \overline{h}', \overline{p}) \sim (\overline{v}, \overline{h}, \overline{p}) \cdot \mathbf{last}.\overline{h}'\} \tag{2.41}$$

A key assumption here is that the *shadow* knows the set of all possible values of h. Using a language-based approach, [109] added H, the *shadow* of the hidden variable h, to the language semantics in much similar way as Joshi and Leino used the so-called "*Havoc on H*", denoted HH, in [80]. Thus, given two states (v_1, h_1, H_1) and (v_2, h_2, H_2), the latter is considered a refinement of the former iff they agree on their v, h-components and the latter's ignorance of h is *at least as much as* the former's. This is formally captured as:

$$(v_1, h_1, H_1) \sqsubseteq (v_2, h_2, H_2) \ \triangleq \ v_1 = v_2 \wedge h_1 = h_2 \wedge H_1 \subseteq H_2$$

Notice the intuition behind the last conjunct in the definition above (i.e., $H_1 \subseteq H_2$) is that *the larger the shadow of the variable h, the more difficult it becomes to learn the value of h in the program.* Lifting this notion of refinement to sets of states S_1, S_2, Morgan concluded that S_2 is an ignorance preserving refinement of S_1 whenever every state $s_2 \in S_2$ is a refinement of some state $s_1 \in S_1$, i.e.:

$$S_1 \sqsubseteq S_2 \ \triangleq \ (\forall s_2 \in S_2 \cdot (\exists s_1 \in S_1 \cdot s_1 \sqsubseteq s_2)) \tag{2.42}$$

In effect, the Formula 2.42 holds that $S_1 \sqsubseteq S_2$ whenever it is that for each initial (v, h, H), every possible outcome (v_2, h_2, H_2) in S_2 satisfies $v_1 = v_2 \wedge h_1 = h_2 \wedge H_1 \subseteq H_2$ for some outcome (v_1, h_1, H_1) of S_1.

In the following section, we discuss the benefits and limitations of existing confidentiality-preserving refinement frameworks.

2.3.5.5 Existing Confidentiality-Preserving Refinement Frameworks - Benefits and Limitations.

It is worth noting that Mantel [99] shed much light on how the notorious problem of the refinement paradox could be resolved. However, since definitions presented in [99], are based on deletion of traces, it follows that the flows actually occurred, but are removed to prevent the adversaries from learning they occurred. We reckon it is much better to prevent insecure flows in the first place.

Jan Jürjens in [82] pointed out one of the limitations of definitions of secrecy properties based on equivalences. The notion of equivalence-based security properties may be summed up as follows. Given a specification P parameterised over a secret variable x, P is said to preserve the secrecy of x if for any change in the value of x, e.g. from x_0 to x_1, the resulting processes $P(x_0)$ and $P(x_1)$ are indistinguishable to any adversary. While secrecy properties of this type yield a high degree of security, [82], argued that for them to be preserved by the traditional refinement relation, they require a rather fine-grained model. Another limitation pointed out by the author is that dropping nondeterministic components during refinement may break the equivalence relation on which such security properties depend, resulting in the well-known problem of *refinement paradox*.

To overcome this propensity towards a refinement paradox, a number of authors in the literature proposed a differentiation between underspecification and unpredictability [96], [108], [109], [82], [129], and then a refining away of underspecification, while preserving *property* unpredictability. Many authors who followed this approach provide different theoretic frameworks for modifying the traditional refinement relation, while preserving unpredictability with respect to the adversary's view of hidden or secret aspects of a specification. What is not very clear, however, is how these various frameworks can be implemented without introducing new languages,

but using existing specification and programming languages. We believe a flow-logic approach for analysing information flow between program parts within well-established languages such as the B Method, could be employed, without necessarily redefining the existing notion of nondeterminism in the industry.

Jüjens pointed out a limitation of stepwise development of secure systems where no distinction is made between underspecification and unpredictability, namely: the need to "redo security proofs at each refinement step"[82]. The author then proposed an approach that involve the use of cryptographic operations to overcome this problem. A prerequisite for this proposed framework is the author's proposal of complete removal of the capability for providing unpredictability through underspecification from the formal model. We do not subscribe to this idea. For one thing, many specification and programming languages in the industry do not make a distinction between underspecification and unpredictability. Hence rather than overhaul the whole method of specifying nondeterminism in the industry, we believe the problem of the intensive labour involved in checking the security proofs at every refinement step can be surmounted by automating the process via static analysis. One advantage of this approach is that it allows for the development of automated tools for analysing information flow that can easily plug into existing CASE tools. What is more, this approach that we subscribe to is simpler and do not involve the use of complicated cryptographic operations.

A limitation of [82], even as acknowledged by the author himself and as remarked by Alur et al [6], is that the framework *may not prevent implicit information flow*. The framework ignores information flow leaks, i.e. cases when the adversary can infer something about the secret (either via termination behaviour or other implicit flows) without explicitly seeing it [6]. The approach we present in chapter 3 of this thesis is more fine-grained, *termination-sensitive* and able to deal with some other *implicit* flows as well. Seehusen and Stølen in [129] acknowledged that while their approach is similar to Jürjens' [82], their formalisms differ. They claimed that while Jürjens distinguishes between underspecification and unpredictability in their definition of secrecy-preserving refinement, he does not rely on this distinction

in the definition of his information flow property. While [129] claimed there is no need to propose conditions with which to check that a given refinement preserve security, they failed to provide tangible examples to illustrate how their proposed secrecy-preserving refinement framework can be employed in practice. One would have hoped to see how they propose to implement the preservation of distinct unpredictability employed in maintaining secrecy, while refining away underspecification. They also acknowledged that they considered only one notion of refinement, behaviour refinement.

Morgan's work in [108], [109] helps recognise the need to formalise information flow security under the assumption that the adversary has strong capabilities including *perfect recall* of program steps that have occurred and the visible variable values after each step; ability to observe program flow and knowledge of program text. To our knowledge the author is one of the few who employed a language-based approach, albeit his model is limited to a basic sequential programming language. The author did not consider, for example loops, and thus possible divergence (or nontermination) and it's effects on information flow security. The author also specialised severely to two basic security levels: *hidden* and *visible*, hence did not consider multi-level security lattices. Like Banks and Jacob in [19] where the notion of *cover stories* is used to preserve unpredictability, [109] added *shadow* variables to the language semantics in a somewhat similar way to what Joshi and Leino [80] did with their notion of *havoc on h*. While notions of security based on definitions of security such as these clouds the adversary's knowledge so he cannot determine, to a high level of certainty, that some secret activity has taken place, it does not actually prevent secret information from flowing to an adversary. Hence [109] acknowledged that they lay no claim to "absolute" security of a program by means of their framework. To our knowledge, it remains to be shown how distinguishing between underspecification and unpredictability, a crux of the secrecy-preserving refinement definitions in [19], [108], [109], [129] and [82], is to be implemented in real specification and programming languages in practice.

Alur et al in [6] argued that an advantage of their framework over Mantel's [99], is that while Mantel assumes some fixed, strong information-flow properties are enforced and his notion of refinement preserves those properties,

their approach is more flexible in that it permits the specification of arbitrary secrecy requirements. Thus [6] claimed that if a specification does not maintain secrecy of a certain property, the implementation does not need to either. Following the stepwise development in our framework, we argue there is no need to implement (or refine) a specification that does not satisfy the security property of interest in the first place. Our automatic information flow analyser will flag the specification as insecure, needing modification before the developer can proceed to the next step in the development process.

2.4 Introduction to the B-Method

Apart, perhaps, from the most elementary software programs, most systems are so large and complex that to be able to fully understand and/or guarantee the consistency, security and other desirable properties of the information manipulated, a system is usually broken into structured components that when synthesised yields the desired functionalities. Thus, breaking the system down into smaller components greatly simplifies the problem domain. This "*divide and conquer*" approach makes the task of formally specifying software systems, using for example the B Method, much more manageable.

The *specification* of the required functionalities of a software system is defined as an *abstract mathematical model*, which acts as though it were a hardware machine carrying out the specified tasks. Hence such models are termed *Abstract Machines*, and like real hardware machines they are comprised of *states* and *operations*. The format and representation used to describe such machines is known as *Abstract Machine Notation* (AMN). The *operations* part of an abstract machine modifies the machine state by means of *Generalised Substitutions*, which are like the engines of the abstract machine in that they define ("*do*") the actual computations of the system. The *Generalised Substitution Language (GSL)*, introduced by Abrial in [1], is the method used to describe and 'assign meaning to' generalised substitutions based on Dijkstra's weakest precondition semantics. The following section deals with the structures or clauses used to build an AMN.

2.4.1 Abstract Machine Notation (AMN) structures

An abstract machine, like any *hardware* machine is itself made up of various parts, namely:

- **MACHINE** - a clause used to specify the machine's identifier, so it could be referenced by other components of the system;
- **PARAMETERS** - (optional) provided with the machine where the user is required to provide some input on which the machine will work. Specifying a machine with parameters makes it more general than it otherwise would be. In line with convention, parameters cannot be changed within a machine, i.e., they can only appear on the right hand side of an assignment operator. Parameters could be finite elementary values or sets of values, which, by default, must not be empty.
- **CONSTRAINTS** - In the case where parameters are provided with a machine specification, the CONSTRAINTS keyword is mandatory, since it is used to provide, as the name implies, constraints on the values that the machine accepts as parameters, i.e., the data types of the parameters.
- **VARIABLES** - provide all the variables to be manipulated locally by the machine, and sometimes, by another machine (via the native machine's operations). The set of values of the variables in a particular instance of the machine defines the 'state' of the machine in that instance.
- **INVARIANT** - gives such mandatory unchangeable information about the machine variables as their data types (e.g., set of natural numbers) and other possible constraints on their values. The invariant must always hold for any value assigned to the variables to be acceptable.
- **INITIALISATION** - mandatory clause used to describe the initial state of the machine by assigning initial values to all variables, such that the invariant is preserved. This guarantees that the machine is feasible, i.e., there is always some state in which the machine works.
- **SETS** - as the name implies is used to describe a collection of scalar values or constants or other structured types that satisfies certain logical properties, which are appropriately specified in the PROPERTIES

clause. Examples include set of positive natural numbers, $NAT1$, set of numbers between 0 and 100, written in ASCII[24] notation as 0..100.

- **CONSTANTS** - Like parameters, constants are read-only entities used within the operations of an AMN and these could be elementary values or structured types such as a maplet[25] of variables to values.

- **PROPERTIES** - this keyword is used to describe the types and logical properties exhibited by constants and sets defined in the machine or other machines accessible to the machine. Machine parameters may also be referenced here. An important point to note here is that for *consistency* of the context of the machine, it must be the case that there are some sets and constants that satisfy the PROPERTIES of the machine, for all possible values of the PARAMETERS [128].

- **OPERATIONS** - provide a list of substitutions that can be performed by the machine or component, along with a precise description of what they each do. A machine interacts with its environment through its operations, collectively referred to as its interface.

Apart from the basic parts of an abstract machine enumerated above, some structures may be added to describe the visibility and/or compositionality of the machines in a development. These structuring mechanisms include: SEES, USES, INCLUDES, EXTENDS, IMPORTS, ... and their functionalities are explained below.

SEES: This clause allows machines in which it is declared (the seeing machines) to have limited shared read-only access to the variables, sets and constants of the seen machine(s). Hence, the *seeing* machines are not allowed to put additional constraints on the variables and constants of the *seen* machine(s) - i.e., *seen* variables can only be used in operations of the seeing machines, and on the right hand side of simple substitutions, but not in the invariant. Because SEES grants *read-only* access, only *enquiry operations* are visible in seeing machines.

[24] ASCII denotes American Standard Code for Information Interchange

[25] Maplet - an ordered pair, i.e., a pair of identifiers ordered by some property or relation that binds them together.

USES: In addition to the access granted by the SEES clause, the USES clause allows the variables of the *used* machine to be used in the invariant of *using* machines. Nevertheless, the USES clause also grants *read-only* access to variables of the used machine, so only *enquiry operations* of the used machine are visible in the *using* machines.

NOTE: To avoid access blocking and inconsistency, update operations of accessed machines are not visible to accessing machines with only shared read access to the variables, sets and constants of the accessed machines. Furthermore both SEES and USES clauses are not transitive, i.e., given three machines denoted $Mach1$, $Mach2$, $Mach3$ and using $\lessdot$ to denote either $SEES$ or $USES$ clauses, whereby $Mach1 \lessdot Mach2$ means $Mach1$ $SEES$ $Mach2$ or $Mach1$ $USES$ $Mach2$, we state formally that:

$$Mach1 \lessdot Mach2 \; and \; Mach2 \lessdot Mach3 \nRightarrow Mach1 \lessdot Mach3$$

INCLUDES: the key difference between the two preceding clauses and INCLUDES is that the latter grants the *including* machine exclusive access to the variables, sets and constants of the *included* machine - hence, one machine cannot be included by two or more machines, i.e., no shared access is permitted. Hence, all operations of the *included* machine, while not automatically becoming operations of the *including* machine, could be used in the *including* machine.

PROMOTES Where some of the operations of an included machine are to be promoted to become operations of the including machine, the PROMOTES clause is declared, following an INCLUDES clause. This clause (PROMOTES) lists all the operations to be thus promoted.

EXTENDS: this clause goes one step further than the INCLUDES / PROMOTES clause combination in that all (rather than some) operations of the *extended* machine are automatically promoted to become operations of the *extending* machine.

IMPORTS: this keyword/clause, *used only in implementations* roughly maps to the INCLUDES keyword *used only in machine specifications*. How-

ever unlike the INCLUDES, USES and EXTENDS clauses, the IMPORTS clause provides full hiding for the abstract variables of the *imported* machine in that it does not allow even a read-only access to these variables in operations of the *importing* machine.

It is noteworthy, however, that while some of these existing structures may be skillfully used to provide a measure of security through partial or full-hiding mechanisms or by *obscurity* via underspecification, the absence of a clause for specifying confidentiality of secret system variables and operations makes it difficult to guarantee or prove that these properties are preserved through refinement.

2.4.1.1 AMN in Refinements

Much has been said about refinements in Section 2.3.5 above, so in this section we will only focus on the *linking* constraints imposed on refinements by the B AMN. As expected the format of refinement is similar to that of the abstract machine specification except for a few structural differences enumerated below:

- Rather than using the MACHINE keyword, a refinement uses the REFINEMENT keyword followed by an identifier, which is immediately followed by a new keyword (not used in abstract machine specifications) REFINES, which specifies the name of machine or intermediate refinement being refined;
- The identifier in the refinement header must be different from that of the system being refined. For example, if the machine to be refined has the header "MACHINE MyMachine", then the refinement header could be "REFINEMENT MyMachineR", but cannot be "REFINEMENT MyMachine";
- While machines could employ composition structures like SEES, USES, INCLUDES or EXTENDS, refinements are not allowed to use the USES clause. This follows from the semantics that a refinement is not allowed to modify the state invariant of a machine but only the internal design of the operations and the introduction of further constraints on the context predicates. For this reason a machine with the

USES clause cannot be independently refined, but must be included along with the *using* machine in another machine, which can then be refined;

- Whereas a machine specification may involve parameters, such parameters must by omitted in the refinement, and the variables of the machine being refined can only be used on the right hand side of a generalised substitution, i.e., they are read only;
- In addition to an AMN refinement being able to use the parallel composition operator, '‖', allowed at the machine level, a refinement can also employ the sequential composition operator, ';', not allowed in a machine.

Another important point to note about AMN refinements is that a refinement can only refine one machine (or intermediate refinement). For example,

$$REFINEMENT\ MyMachineR$$
$$REFINES\ MyMachine$$

is well-formed, whereas

$$REFINEMENT\ MyMachineR$$
$$REFINES\ MyMachine1,\ MyMachine2$$

is not allowed.

2.4.1.2 AMN in Implementations

An AMN implementation is the last step in the step-wise refinement process in B, and as in the case of machines and refinements, it starts with a header, IMPLEMENTATION, and follows with a REFINES clause as in a refinement. The general form of an AMN implementation, abbreviated, is:

$$IMPLEMENTATION\ MyMachineI$$
$$REFINES\ MyMachineR$$
$$\ldots$$

As in the case of refinements, there are some structural differences between an implementation and a machine, yes, even some differences between an

implementation and a refinement as itemised below:

- Unlike machine and refinement components where the parallel composition operator is allowed, the only composition operator allowed in an implementation is the sequential composition operator. This follows from the fact that an implementation must provide the needed functionality for the system, and no further refinement is possible afterward. And for the same reason ...;
- WHILE loops may only be used in implementation. Loops are not allowed in abstract machine specifications and refinements. The LET keyword on the other hand may only be used in abstract machines and refinements. It is not allowed in implementations, but the VAR keyword is.
- Local sets that up till now could be abstractly defined (aka *deferred sets*) must be enumerated, i.e. fully defined in terms of its elements via the **VALUES** clause. And similarly all constants of the refined component must be precisely defined, hence constants of structured types such as a maplet of variables to values are not allowed in an implementation. Rather, only constants of elementary values are allowed;
- The only structuring mechanism common to all three is the SEES clause. Neither the INCLUDES nor the EXTENDS clauses are allowed in an implementation. Rather, the IMPORTS is used here and this corresponds to the INCLUDES clause in abstract machine specifications and refinements.
- The VARIABLES clause is not allowed in implementations, except for concrete variables; hence if (abstract) variables defined in refined components are to be used in an implementation, such machines must be imported into the implementation using the IMPORTS clause. Following from that, it is not mandatory to have the INITIALISATION clause as is the case in machines and refinements.

We summarise the key points presented in Section 2.4.1 in the table in Figure 2.15.

Description		*AMN Components*		
		Specification	*Refinement*	*Implementation*
Keyword / Clause	MACHINE	●		
	REFINEMENT		●	
	IMPLEMENTATION			●
	REFINES		●	●
	PARAMETERS	●		
	CONSTRAINTS	●		
	SETS	●	●	● (Enumerated)
	CONSTANTS	●	●	● (Elementary)
	PROPERTIES	●	●	● (Fully defined)
	VARIABLES	●	●	
	CONCRETE VARIABLES		●	●
	INVARIANT	●	●	● (Optional)
	INITIALISATION	●	●	● (Optional)
	OPERATIONS	●	●	●
Structuring Mechs	SEES	●	●	●
	USES	●		
	INCLUDES	●		
	EXTENDS	●		
	IMPORTS			●
	PROMOTES			●

Figure 2.15: Summary of the B AMN Clauses and their Use

2.4.2 Refinement Proof Obligations in the B Method

For a machine and any subsequent refinement thereof to satisfy the requirements for which they are designed, they have to meet a number of proof obligations. Hence, we briefly discuss the proof obligations for standalone B Machines and their refinements in this section. (A more detailed discussion of this concept by Abrial can be found in [1].) Suppose we have a machine $M(p)$ described as follows, where p ranges over parameters, *constr* denotes constraints, ST_M ranges over sets, k_M are constants, T_M denotes properties, v_M ranges over variables, I_M, $init_M$ are invariants and variable initialisations respectively, xx, aa are output and input variables respectively, P_M is a predicate, and *subst* denotes the body of the generalised substitution:

MACHINE $M(p)$
CONSTRAINTS *constr*
SETS ST_M
CONSTANTS k_M
PROPERTIES T_M
VARIABLES v_M
INVARIANT I_M
INITIALISATION $init_M$
OPERATIONS
$xx \longleftarrow oper(aa) \triangleq$
PRE P_M **THEN**
subst
END
END

The first crucial proof obligation in a B machine is to prove that there is at least one valid state that satisfies the machine invariant, thus indicating that the machine is feasible. This minimal feasibility condition requires that the machine is executable in its initial state defined in the initialisation clause. For $M(p)$, this obligation is:

$$constr \ \wedge \ T_M \ \Rightarrow \ [init_M]I_M$$

Notice that $init_M$ is guaranteed to preserve I_M only in the context where the parameters satisfy the constraints *constr* and the sets and constants satisfy the property T_M. The next important obligation we need to prove for machine $M(p)$ is that the body of the operation $oper(aa)$ establishes the invariant I_M. This obligation extends the context of the preceding one by adding the requirement that the precondition P_M holds for *subst* to establish I_M as shown below:

$$constr \ \wedge \ T_M \ \wedge \ P_M \ \Rightarrow \ [subst]I_M$$

Given that $MR(p)$, described below, is a refinement of $M(p)$, we can now begin to explore the B refinement proof obligation for the development. Here ST_R, k_R denote sets and constants respectively; T_R denotes properties defined on the sets and constants; v_R, I_R denote variables and invariant re-

spectively, while $init_R$ denotes substitutions used to initialise variables; P_R is a predicate and $subst_R$ is the body of the substitution defined in $oper(aa)$ while $yy,\ aa$ range over output and input variables respectively.

REFINEMENT $MR(p)$
REFINES M
SETS ST_R
CONSTANTS k_R
PROPERTIES T_R
VARIABLES v_R
INVARIANT I_R
INITIALISATION $init_R$
OPERATIONS
$yy \longleftarrow oper(aa) \ \triangleq$
PRE P_R **THEN**
$subst_R$
END
END

The first crucial refinement proof obligation for $MR(p)$ is that the local states of the refinement relates to some states of the machine being refined [1], [68]. This describes the *refinement* requirement that some states of the refinement are also valid states of the original machine, i.e.:

$$constr \ \wedge \ T_M \ \wedge \ T_R \ \Rightarrow \ \exists(v_M, v_R) \cdot (I_M \wedge I_R)$$

Following from this, we need to show that the initialisation $init_R$ of $MR(p)$ refines the initialisation $init_M$ of $M(p)$, i.e.:

$$constr \ \wedge \ T_M \ \wedge \ T_R \ \Rightarrow \ [init_R]\neg[init_M]\neg I_R$$

This initialisation obligation requires that: whenever the initialisation of the machine, $init_M$, establishes the invariant of the refinement, I_R, it must be the case that the initialisation $init_R$ of the refinement also does. Thus $init_R$ is a refinement of $init_M$.

Similarly, the proof obligation for the operation of the refinement requires that every possible execution of the refinement operation correspond to some valid execution of the related machine operation, i.e.:

$$constr \ \wedge \ T_M \ \wedge \ T_R \ \wedge \ I_M \ \wedge \ I_R \ \wedge \ P_M \\ \Rightarrow \ P_R \ \wedge \ [subst_R]\neg[subst]\neg(I_R \wedge xx = yy)$$

Having discussed Refinement Proof Obligations in the B Method, we present in Section 2.4.3 situations whereby a machine and its refinement, while satisfying necessary refinement proof obligations, may leak some information between the variables of the machine.

2.4.3 Confidentiality Refinement Paradox in the B Method

In this section, we present two examples showing that the use of a formal specification method in itself does not automatically resolve the problem of confidentiality refinement paradox discussed earlier in Section 2.3.5.2.

The idea behind the first example is to specify a system that records student registration numbers, coursework marks and exam marks. However, the lecturer (the power user) desires to permit his teaching assistant to enter and read only coursework marks, while preventing him from learning the exam marks of the students. Thus, the state variable holding the exam marks ($exam\_H$) is classified as secret or *high-security*, whereas the state variable holding the coursework marks (cw_L) is classified public or *low-security*. The Abstract Machine specifying this requirement, *StudentRecords*, is given in Table 2.6.

Table 2.6: Abstract Machine - StudentRecords.

****

MACHINE

$StudentRecords(maxInteger, maxMark)$

CONSTRAINTS

$maxInteger \in \mathbf{NAT1} \ \wedge \ maxMark \in \mathbf{NAT1}$

Continued on next page

Table 2.6 – continued from previous page

SETS

$REG\ ;\ MARK\_RANGE$

CONSTANTS

$maxStudents$

PROPERTIES

$maxStudents \in NAT1$

VARIABLES

$stId,\ exam\_H,\ cw\_L,\ cwMarkPair,$
$examMarkPair$

INVARIANT

$stId \in \mathbf{NAT1} \ \wedge \ stId <= maxInteger \ \wedge$
$exam\_H \in \mathbf{NAT} \ \wedge \ exam\_H <= maxMark \ \wedge$
$cw\_L \in \mathbf{NAT} \ \wedge \ cw\_L \leq maxMark \ \wedge$
$cwMarkPair \in \mathbf{NAT1} \rightarrow \mathbf{NAT} \ \wedge$
$examMarkPair \in \mathbf{NAT1} \rightarrow \mathbf{NAT}$

INITIALISATION

$stId,\ exam\_H,\ cw\_L,\ stMarkPair :=$
$1,\ 0,\ 0,\ \{\},\ \{\}$

OPERATIONS

$mm \longleftarrow update\_Cw(regNum, mark) \ =$
PRE
$regNum \in \mathbf{NAT1} \ \wedge$
$regNum <= maxInteger \ \wedge$
$mark \in \mathbf{NAT} \ \wedge \ mark <= maxMark$
THEN
$cwMarkPair(regNum) := mark \ \|$
$mm := mark$
END;
$nn \longleftarrow update\_Exam \ =$
ANY $ss,\ hh$ **WHERE**
$ss \in \mathbf{NAT1} \ \wedge \ hh \in \mathbf{NAT} \ \wedge$
$hh <= maxMark$
THEN

Continued on next page

Table 2.6 – continued from previous page

```
                SELECT
                    ss <= maxInteger
                THEN
                    exam_H := hh  ||
                    examMarkPair(ss) := hh  ||
                    nn := exam_H
                ELSE
                    exam_H := 0  ||  nn := 0
                END
            END
        END
```

Notice in Table 2.6 that the high security variable $exam\_H$ is updated with an arbitrary value between 0 and $maxMark$ within the operation $nn \longleftarrow update\_Exam$. Similarly, the low security variable cw_L is updated with any value within the same range by the operation $mm \longleftarrow update\_cw$. Notice also that due to the nondeterminism introduced by the underspecification (with the ANY clause) in $nn \longleftarrow update\_Exam$, it is not possible for cw_L to learn the initial value arbitrarily assigned to $exam\_H$. Hence, the machine trivially satisfies GNI.

However, we give, in Table 2.7, a refinement, which while itself is a valid refinement by the classical refinement semantics, makes the initial value of **exam_H_r** flow implicitly into **cw_L_r** within the operation **mm ⟵ update_Cw** (due to the precondition $mark = exam\_H\_r$) thereby making the refinement fail to satisfy NI. (We highlight in red the substitutions and constraints that yield the insecure flow.) Yet, by the semantics of B GSL the refinement is valid, and the insecure implicit flow of information from **exam_H_r** to **cw_L_r** escapes detection by tools like the B Toolkit, Atelier B, etc. Figure 2.16 shows that both machine and refinement satisfy all required proof obligations and commit successfully in Atelier B. This example clearly illustrates that rigorous formalism does not in itself guarantee

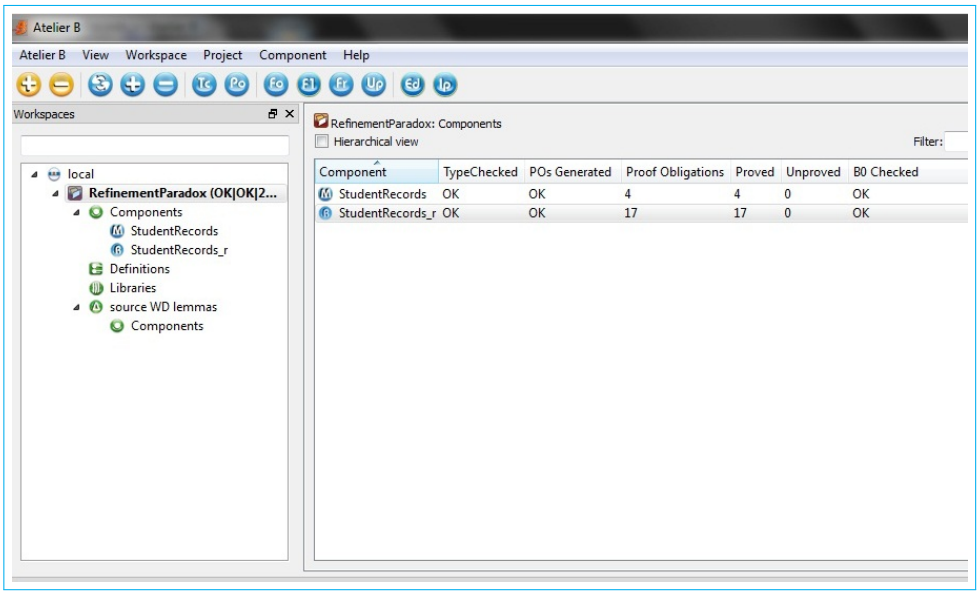


Figure 2.16: Atelier B PO Screen for StudentRecords

information flow security.

Table 2.7: Demonic Refinement - $StudentRecords\_r$.

****

REFINEMENT

$StudentRecords\_r(maxInteger,\ maxMark)$

REFINES

$StudentRecords$

VARIABLES

$stId\_r,\ exam\_H\_r,\ cw\_L\_r,\ cwMarkPair\_r,$
$examMarkPair\_r$

INVARIANT

$cw\_L\_r \in \mathbf{NAT} \ \wedge \ stId\_r \in \mathbf{NAT1} \ \wedge \ exam\_H\_r \in \mathbf{NAT} \ \wedge$
$stId\_r = stId \ \wedge \ exam\_H\_r = exam\_H \ \wedge \ cw\_L\_r = cw\_L \ \wedge$
$\mathbf{cw\_L\_r = exam\_H\_r} \ \wedge \ cwMarkPair\_r = cwMarkPair \ \wedge$
$examMarkPair\_r = examMarkPair$

Continued on next page

Table 2.7 – continued from previous page

INITIALISATION

$stId\_r,\ exam\_H\_r,\ cw\_L\_r,\ cwMarkPair\_r,$
$examMarkPair\_r\ :=\ 1,0,0,\{\},\{\}$

OPERATIONS

$mm \longleftarrow update\_Cw(regNum,\ mark)\ =$
PRE
$regNum \in \mathbf{NAT1}\ \wedge\ regNum <= maxInteger\ \wedge$
$mark \in \mathbf{NAT}\ \wedge\ mark <= maxMark\ \wedge$
mark = exam_H_r
THEN
cw_L_r := mark ;
$cwMarkPair\_r(regNum) := mark;$
$mm := mark$
END;
$nn \longleftarrow update\_Exam\ =$
ANY ss , hh **WHERE**
$ss \in \mathbf{NAT1}\ \wedge\ hh \in \mathbf{NAT}\ \wedge\ hh \leq maxMark\ \wedge$
$stdId\_r = ss\ \wedge\ exam\_H\_r = hh$
THEN
IF
$ss <= maxInteger$
THEN
$exam\_H\_r := hh\ \ \|$
$nn := examMarkPair\_r(ss) := hh$
END $\|\ nn := hh$
END
END

We next present a second example showing that formalism in itself is not sufficient to assure secure information flow between program variables. Here we write h, l to denote *secret* and *public* boolean variables respectively, with 0 denoting the value '*false*' and 1 denoting '*true*'. We then construct three

operations, S_1, S_2, and S_3 as follows:

$$S_1 \triangleq \textbf{CHOICE } h := 1 \textbf{ OR } h := 0 \textbf{ END}$$
$$S_2 \triangleq \textbf{CHOICE } l := 1 \textbf{ OR } l := 0 \textbf{ END}$$
$$S_3 \triangleq \textbf{BEGIN } l := h \textbf{ END}$$

We illustrate all possible transitions of S_2 in Figure 2.17, wherein we highlight transitions that correspond to all possible transitions of S_3 (compare with Figure 2.18).

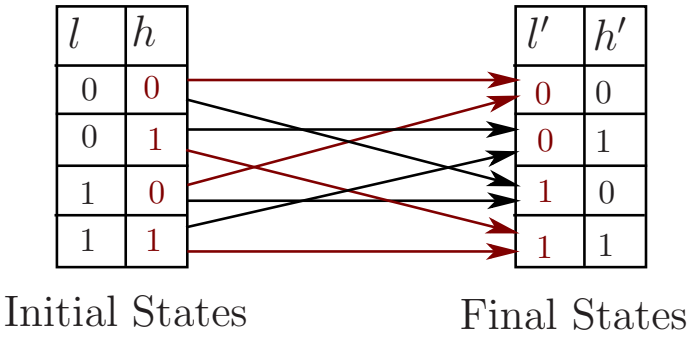


Figure 2.17: Possible transitions of S_2

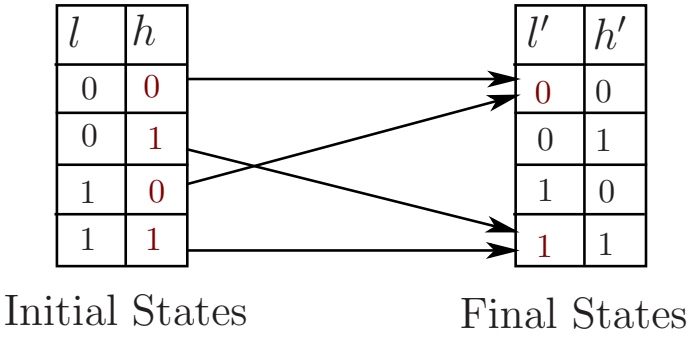


Figure 2.18: Possible transitions of S_3

Notice that Figure 2.17 illustrates the set of transitions:

$$\{(0,0) \to (0,0),\ (0,0) \to (1,0),\ (0,1) \to (0,1),\ (0,1) \to (1,1),$$
$$(1,0) \to (1,0),\ (1,0) \to (0,0),\ (1,1) \to (0,1),\ (1,1) \to (1,1)\},$$

whereas Figure 2.18 illustrates the set of transitions:

$$\{(0,0) \rightarrow (0,0),\ (0,1) \rightarrow (1,1),\ (1,0) \rightarrow (0,0),\ (1,1) \rightarrow (1,1)\}.$$

Thus, in the presence of underspecification, both S_1 and S_2 trivially satisfy GNI. For example, from the transitions of S_2 illustrated in Figure 2.17, an adversary is unable to conclusively deduce whether information flows from h to l or not. When the underspecification in S_2 is refined away, however, we get S_3 (transitions illustrated in Figure 2.18), which is admissible by the semantics of classical refinement. It is clear from the foregoing that all the transitions in S_3 are possible transitions of S_2. Yet, although being a valid refinement of S_2, S_3 is not *secure*, as it allows the initial value of the secret variable h to explicitly *interfere* with the final value, l', of the public variable l. Also, a comparison of Figures 2.17 and 2.18 shows that S_2 *masks* the possibility of insecure flows, which is exposed when the underspecification in S_2 is refined away as in S_3. Hence the two examples presented in this section show that the use of a formal method in itself does not guarantee secure information flow through refinement. This observation corroborates the point noted earlier in Sections 2.3.5.1 and 2.3.5.4 about the lack of distinction between nondeterminism used for underspecification and nondeterminism used for unpredictability in traditional refinement frameworks.

In the following subsection, we review some existing work wherein formal developments using the B Method is imbued with some notions of confidentiality.

2.4.4 Existing Security Frameworks using The B Refinement Process

A number of authors have introduced frameworks for modeling security properties using the B refinement process in the areas of computer networks [120], authentication protocols [24], [25], contractual obligations and data sharing agreements [9], [14], [10] and [11]. Since many of the authors used Event-B, we give a brief introduction to the Event-B method in Subsection 2.4.4.1 and thereafter we precís some of the security frameworks using the B refinement process.

2.4.4.1 A Brief Introduction to Event-B

Event-B is a simplification and extension of the B Method whereby abstract machines and refinements are described by events, instead of operations. Another major difference between the B Method and Event-B is that Event-B is specialised for action systems whereas the B Method is a general-purpose formal development method. The Event-B method is semantically a simple or *guarded* discrete transition system using the language of classical logic and set theory whereby *states*, represented as valuations of *sets of variables*, are manipulated by actions defined in *events* which are described by *generalised substitutions* [2], [3], [119], [120], [9]. Formal developments in Event-B are based on the concept of *models* constituted of two types of components, namely: *machines* and *contexts*. An Event-B machine describes the model's dynamic behaviour by means of *state* and events, while Event-B contexts define constants that are either numeric or sets [124], [2]. A machine may see one or more contexts, and whereas a machine may be *refined*, a context may only be *extended*.

Event-B allows three kinds of substitutions namely: the *empty* substitution, the *deterministic* substitution, and the *nondeterministic* substitution [3]. Given that x denotes a list of variables ranging over **Ide**; $E(\mathbf{Ide})$ denotes a number of set-theoretic expressions corresponding to each of the variables in x in the deterministic case, while, in the nondeterministic case, t denotes a collection of distinct fresh variables which are local to the generalised substitution; $P(t, \mathbf{Ide})$ denotes a conjoined list of predicates, and $F(t, \mathbf{Ide})$ denotes a number of set-theoretic expressions corresponding to each of the variables in x. We summarise the kinds of generalised substitutions for expressing the transitions associated with events in EBNF notation below:

$$
\begin{array}{lll}
S \triangleq \mathbf{SKIP} & & \text{- empty substitution} \\
\quad \mid x \coloneqq E(\mathbf{Ide}) & & \text{- deterministic substitution} \\
\quad \mid \mathbf{ANY}\ t\ \mathbf{WHERE} & & \text{- nondeterministic} \\
\qquad P(t, \mathbf{Ide})\ \mathbf{THEN} & & \text{substitution} \\
\qquad F(t, \mathbf{Ide})\ \mathbf{END} & & \\
\quad \mid S_1 \parallel S_2 & & \text{- parallel substitution}
\end{array}
$$

Thus given that G denotes a guard defined as a formula in classical first

order logic, S denotes a generalised substitution, e ranges over events, an event model, ev, is defined as:

$$ev \triangleq \textbf{EVENT } e \textbf{ WHEN } G \textbf{ THEN } S \textbf{ END}$$

2.4.4.2 Security Frameworks of Interest

With respect to the enforcement of security policies in networks, Stouls and Potet in [120] argued that rather than the common separation between policies and the mechanisms for their implementation, a formal link can be built between the abstract and concrete systems that integrates the security policy of interest into the mechanism for its implementation. Thus they proposed that specifications be pitched at the same level of abstraction as security policies and then refined down to the concrete mechanism for enforcing the policies, which the authors then built into the Event-B models. The authors modeled network communications using the TCP/IP protocol suite[26], with sub-protocols for communicating between corresponding layers in the protocol stack. Traditionally, network security policies are defined in terms of access control rights whereby access policies are defined on sets of actions by a closed set of rules. Policies are either defined as forbidden actions (*negative authorisations*), aka. open policies, or authorised actions (*positive authorisations*), aka closed policies, or a combination of both [120].

As a follow-up to [25], Bieber and Boulahia-Cuppens in [24] introduced an application of the B Method to the formal development of authentication protocols and a systematic refinement of the protocols from abstract to concrete. The authors followed a top-down (rather than the commonly used bottom-up) approach in their study of authentication protocols, starting with a very abstract specification, which they progressively refined to implementation. This necessitates the abstraction of authentication primitives[27], the logic of which is embedded within the INVARIANT clause of B

[26]The TCP/IP (Transmission Control Protocol and Internet Protocol) suite is a four level protocol stack consisting of the *Application, Transport, Network*, and *Link* (or *Network Access*) layers.

[27]Authentication primitives are atomic elements required to guarantee authentic and secure communication, e.g., 'only authorised participants may send message on communication channel', 'unauthorised players may *not* read message on communication channel',

machines using classical first order logic and set theory. The authors illustrated attestation of identity (i.e. *entity authentication*[28]) [103] to prevent *masquerading* or *forging* and data origin (i.e., *message authentication* [29]) [103], [142], which deals with the prevention of *listening* or *eavesdropping* as well as *blocking* whereby the adversary receives message and prevents intended recipient from receiving it.

The authentication framework introduced by Bieber and Boulahia-Cuppens in [24], [25] used two approaches to implement data origin (message) authentication. The first involves *Envelopes*, which are messages that can be transmitted on a channel with the presence of adversaries. As the name implies, an envelope is modeled with three attributes, namely: *content, address, source*, where content is the clear-text message to be sent securely, the address is the intended receiver's identity and the source is the sender's identity. The other approach involves the modeling of data origin authentication with cryptographic keys rather than envelopes. Here, the sender encrypts the clear-text message before sending it on the channel and the receiver decrypts it with an agreed key.

Aziz et al in [9] introduced a syntactic extension to constrain Event-B events such that they may only execute when defined obligations are met. An event is *permitted* whenever the guard holds in a given state, but the event may or may not execute. The authors [9] introduced a dual of guards, termed *triggers*, which expresses an *obligation* on when permitted events must be executed. The authors did this by putting extra constraints on each of the other permitted events in a nondeterministic setting, prohibiting their immediate execution, i.e., if any event is to execute next, it must be the one *not prohibited* by the trigger. To accomplish this the authors introduced new constructs '**NEXT**', '**EVENTUALLY**', and '**WITHIN** ... **NEXT**' to replace the **THEN** part of the event model, as shown below, where T denotes the trigger, a predicate, and $n \in \mathbf{NAT}$ is a natural number:

$$ev \ \widehat{=}\ \mathbf{EVENT}\ e\ \mathbf{WHEN}\ T\ \mathbf{NEXT}\ S\ \mathbf{END} \tag{2.43}$$

etc.

[28] Entity authentication deals with the verification of an entity's claimed identity [67].

[29] Message Authentication certifies message integrity and source.

$$ev \triangleq \textbf{EVENT } e \textbf{ WHEN } T \textbf{ EVENTUALLY } S \textbf{ END} \tag{2.44}$$

$$ev \triangleq \textbf{EVENT } e \textbf{ WHEN } T \textbf{ WITHIN } n \textbf{ NEXT } S \textbf{ END} \tag{2.45}$$

Model 2.43 indicates that the next event to execute when the trigger T holds is e. Model 2.44 shows that while the number of events that may execute before e is not specified (unbounded nondeterminism), e will *eventually* be executed. Model 2.45 defines bounded nondeterminism meaning that the event e must be executed *within* n number of events. To demonstrate how [9] employed triggers, consider the example of a system with two events e and f where G denotes guard, T denotes trigger, and $S_1,\ S_2$ denotes generalised substitutions:

$ev1 \triangleq \textbf{EVENT } e \textbf{ WHEN } G \textbf{ THEN } S_1 \textbf{ END}$ - regular Event-B

$ev2 \triangleq \textbf{EVENT } f \textbf{ WHEN } T \textbf{ NEXT } S_2 \textbf{ END}$ - modified Event-B

The system specification given above requires that whenever T is true, the next event to be executed must be f. This in turn requires that e be prohibited as shown in the refined specification given below.

$ev1 \triangleq \textbf{EVENT } e \textbf{ WHEN } G \wedge \neg T \textbf{ THEN } S_1 \textbf{ END}$ - regular Event-B

$ev2 \triangleq \textbf{EVENT } f \textbf{ WHEN } T \textbf{ NEXT } S_2 \textbf{ END}$ - modified Event-B

Using the notion of obligations introduced in [9], the authors in [11], [14] introduced an Event-B framework for modeling Data Sharing Agreements (DSA)[30], parameterised on obligation, authorisation and prohibitions. [11] presented DSA clauses as guarded actions, where the guard constitutes the context in which the DSA holds. Writing $p,\ an,\ d$ to denote the principal, an *a*ction *n*ame, and data of interest respectively, the authors defined an action as a tuple $\langle p, an, d\rangle$ indicating that p performs an action, an, on some data, d. DSA clauses are defined by four types. Given that n is an integer and $a = \langle p, an, d\rangle$, the clause types are: permissions (denoted $\mathcal{P}(a)$), prohibitions (denoted $\mathcal{F}(a)$), bounded obligations (denoted $\mathcal{O}_n(a)$), and unbounded obligations (denoted $\mathcal{O}(a)$). Hence [11] defined the syntax of DSA clauses, $\mathcal{C}$ as

[30]A Data Sharing Agreement is a contract designed to regulate the sharing of data among multiple participants or principals in several specific domains and contexts, which is a predicate characterising environment conditions such as location and time [136], [11].

$$\mathcal{C} \triangleq IF\ G\ THEN\ \mathcal{P}(a) \mid IF\ G\ THEN\ \mathcal{F}(a) \mid IF\ G\ THEN\ \mathcal{O}_n(a) \mid IF\ G\ THEN\ \mathcal{O}(a)$$

Bounded obligations are defined in the INVARIANT of the Event-B model using the '$WITHIN \ldots NEXT$' trigger clause introduced in [9], while the guard is defined in the CONTEXT clause of the Event-B model. When a number of events are permitted within a context, any one of them could be nondeterministically selected. It may be the case, though, that two or more events perform conflicting actions, and need to be safely resolved. For example, a system modeling a train station as a shared resource between a train company (denoted TC) and emergency services (denoted ES) may have the following conflicting operational goals:

1. TC: *Always open station between 05.00hrs and 23.59hrs*
2. ES: *In the event of a fire alert, evacuate and close station*

Using the framework introduced in [9], [10], [11] and [14], the station events above can be semi-formally rewritten as follows, given that $fireAlert$ is a predicate (trigger), *time* is a predicate (guard), and *openStation*, *closeStation* are both generalised substitutions:

1. TC : **EVENT** e **WHEN** $time \in 05.00..23.59 \ \wedge \ \neg(fireAlert)$ **THEN** *openStation* **END**
2. ES : **EVENT** f **WHEN** $fireAlert$ **NEXT** *closeStation* **END**

Another scenario where two or more events may perform conflicting actions that require resolution can the seen in the development of a telephone system where *call waiting* and *call forwarding* functionalities are required (when line is busy) as loosely specified below. Here, we assume $lineBusy$ is a boolean, $callWaiting$ is a GSL substitution that specifies that incoming call be kept on hold, and the GSL $forwardTo$(**NAT**) specifies the phone number the incoming call is to be forwarded to.

3. CallWaiting: **EVENT** e **WHEN** $lineBusy$ **THEN** $callWaiting$ **END**
4. CallForwarding: **EVENT** f **WHEN** $lineBusy$ **THEN** $forwardTo(07828123456)$ **END**

Again, following a similar approach to the one used in the train station example, we can resolve the conflict apparent in the specifications 3 and 4 above as shown below, where we assume $waitingTime \in$ **NAT1** and $waitingTime \leq 60$ denotes number of seconds calls may be kept waiting.

3. CallWaiting: **EVENT** e **WHEN** $lineBusy$ **NEXT** $callWaiting$ **END**
4. CallForwarding: **EVENT** f **WHEN** $lineBusy \wedge$
   $waitingTime > 60$ **THEN**
   $forwardTo(07828123456)$ **END**

Aziz et al in [10] introduced a framework for extending Event-B machines to manage conflicts of interest in the context of collaborating virtual organisations[31] in Grid Computing.

While all the frameworks ([120], [24], [25], [9], [10], [11] and [14]) discussed in this section apply some notions of security to software models developed using the B refinement process, none, to our understanding, dealt with the analysis of information flow such as between variables within a development. One thing all the frameworks have in common is that they all introduced modalities for constraining the INVARIANT clause within B models as necessary. At an early stage of the research leading up to this thesis, we considered a similar approach, i.e., extending the B GSL syntax with clauses for checking information flow security, using the security conditions derived from our information flow analysis framework. The reasons we did not develop this intuition further are discussed below:

- Unlike DSAs, which are generally *static* and can readily be represented using linear temporal logic, information flow security is a property of multiple runs of a system and has been shown by Alur et al in [6] as not expressible in linear temporal logic.

- Trying to build information flow constraints into B machines will compound the proof obligations, with the prospect that industry practitioners may not be readily inclined to adopt the approach. This will conflict with one of our motivations for this research work, namely: *simplicity*, as discussed in Section 1.2.

[31] A virtual organisation is a group that shares the same computing resources

- Another reason for not modifying the B language is that most B and Event-B tools are Ecclipse-based, which makes it easy to develop an analyser, external to the core B tool, and then plug it into Ecclipse rather than implement a hardcore approach such as extending the language kernel.

- Furthermore, trying to build information flow constraints into B machines will require extending existing tools like the B Toolkit, Atelier B, proB, etc, to recognise the necessary additional keywords / clauses that would result. Conversely, having the analyser as an external standalone tool that takes B machines as input does not require extending existing tools, and works just as well. Furthermore, legal and copyright issues will be an unwelcome distraction to our task of analysing information flow in B developments if we seek to extend the existing tools mentioned.

- Having our information flow analyser external to B machine models makes it easier to adapt the analyser to deal with models developed using other formal methods. If, on the other hand, information flow constraints are built into individual B machines, the flow analysis framework will be less portable.

Having presented existing work in the literature with respect to information flow security, programs and their refinements, and the B Method, we use this background knowledge to develop an information flow analysis framework in Chapter 3.

Chapter 3

Standalone B Machines and Generalized Noninterference

3.1 Introduction

In Section 2.4, we presented a brief informal introduction to the B Method. In this chapter, we present the formal framework for developing B Machines that satisfy Generalized Noninterference in the presence of *underspecification*, and on implementation can be guaranteed to satisfy, correspondingly, Noninterference. The approach we introduce here involves a program analysis framework for the abstraction and certification of secure information flow within B machines and refinements, *in the presence of nondeterminism* and corresponding B implementations *in the absence of nondeterminism*.

First, we discuss our approach, which involves *a separation of concerns*, whereby, we divorce the problem of preservation of the security properties of interest from the traditional problem of refinement of B machines. We follow this with a definition of the abstract syntax and semantics of the abridged version of the B Generalised Substitution Language (GSL) used in this thesis. Next, we present some of our main contributions in this chapter, namely:

- An information flow analysis of standalone B Machines;
- Introduction of security conditions that ensure the preservation of security properties through refinement; and

- Examples illustrating how our framework could be used to ensure secure information flow within a development.

We show that the syntactic security conditions imply the semantic one derived from the information flow analysis.

Later, in Chapter 4, we discuss the extension of our flow analysis framework to structured developments in B. We thereby develop an analysis framework for information flow in B Machines within a development in the presence of *structuring mechanisms*. In Chapter 5, however, we present a case-study that illustrates our implementation of the flow analysis framework, using C++; and we conclude Chapter 6 with our intuitions on how our information flow analysis framework may be extended further.

3.2 Security Properties and Refinement

In Chapter 2, we discussed the problem of the infamous refinement paradox, and the fact that many approaches (such as Mantel's and Jürjens') that sought to solve this problem by adding constraints to the classical refinement relation have, at best, had limited success [99],[100], [82], [23] (see Section 2.3.5.5). To the best of our knowledge, there is no framework within the B Method designed to deal with the problem of preservation of confidentiality properties through refinement; neither are we aware of any claim that existing tools like the B Toolkit, ProB, etc. deals with this problem.

To overcome this problem and bridge the gap between theory and practice, we decompose the problem of preservation of security properties through refinement into two separate manageable problems: *the problem of refinement* and *the problem of preservation of the security properties of interest.* We use the traditional B development method involving *proof obligations* to check the validity of B machines and their refinements, after which we use the constraints developed using our program analysis framework to *automatically* check the B machines and their refinements for possible insecure flows of information at every step of the B development process. If a B machine, passes both tests, then we conclude that we have a *valid refinement that preserves the security property of interest.*

The main advantage of automating the information flow analysis part of our solution is that it removes the extra human labour that would otherwise be involved in checking for insecure flows at every step of the refinement process, a problem alluded to by Jürjens in [82]. Other reasons motivating against manually carrying out the information flow analysis part of our solution include the increased likelihood of human error; needless delays due to the extra work required; and increased cost due to the need for more technically capable people to manually do the analysis.

We reason that: *if the wheel works, why reinvent it?* Since the classical refinement relation captures other (other than security, that is) desirable properties like safety, we conclude that it is sufficient to enhance the development process with information flow constraints, via our program analysis approach, for example, rather than *overhaul* an already well-known and effective refinement framework. Hence, we use existing tools like the B Toolkit, AtelierB, ProB, etc. to validate B machines and their refinements, while we employ our proposed information flow analysis tool to validate the preservation of defined security policies on the *checked* machines and their refinements. We illustrate our decomposition approach in Figure 3.1

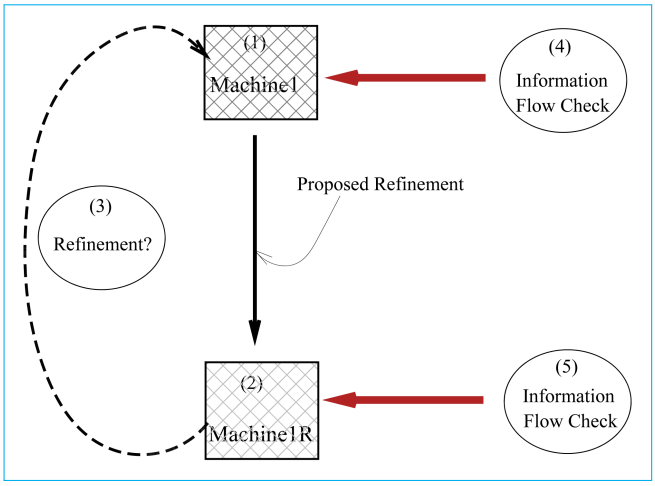


Figure 3.1: Problem decomposition: Information Flow and Refinement

In the following section, we discuss the syntax and semantics of the abridged B GSL on which our information flow analysis framework presented in Section 3.4 is based.

3.3 Abstract Machines and Generalised Substitutions

We present in this section a formal definition of abstract machines; our slightly modified syntax and semantics of generalised substitutions and the basic theories behind the definitions, as introduced by Abrial in [1].

3.3.1 Syntax and Semantics of Generalised Substitutions

We present a formal introduction to the grammatical rules and form or layout (aka. *syntax*) of the B GSL as well as the semantics, i.e., the behaviour of the program based on meanings assigned to the defined syntax. Recall from Section 2.3.4 that $[S]Q$ is an alternative notation for the weakest precondition semantics. And $[S]Q$ collects *the largest set of initial states from which the substitution S is guaranteed to establish the postcondition Q on termination of S*. For example, given that $S \triangleq x := x - 1$ and $Q \triangleq x > 0$, then $[S]Q$ collects the initial set of states from which $x := x - 1$ is guaranteed to establish $x > 0$ on termination, i.e., the set of states where $x > 1$ holds initially. Writing x' to be a *fresh* variable denoting the final value of x after the substitution, we have:

$[x' := x - 1](x' > 0)$

(Substituting $x - 1$ for x' in $(x' > 0)$, we have)

$x - 1 > 0$

i.e., $x > 1$ □

Hence the weakest precondition that $x := x - 1$ satisfies $x > 0$ is $x > 1$.

The notion of simple substitution illustrated above is what Abrial [1] generalised to the formal specification of all abstract machine operations. Before we present the formal syntax and semantics of abstract machine operations, we introduce in Table 3.1 the following notational conventions which are used throughout the rest of this thesis.

Symbol	Meaning
Exp	Finite set of expressions in program
$E \in$ **Exp**	A single expression
Lab	A finite set of labels
$\ell \in$ **Lab**	An individual label
Statement	Set of labeled substitutions in program
$S \in$ **Statement**	A labeled substitution
Subst	A set of substitutions
${}_{a}C \in$ **Subst**	A substitution allowed in MACHINEs only
${}_{\star}C \in$ **Subst**	A substitution allowed in IMPLEMENTATIONs only
$C \in$ **Subst**	A substitution allowed in machines and implementations
P, Q, R	Predicates over state variables
U	A set of defined items
$pred(U)$	Predicate defining set U
Ide	A finite set of variables
$x \in$ **Ide**	An individual variable
b	A boolean expression
$\bullet$	A special metavariable, denotes '*Abort*' possible
$\widehat{\textbf{Ide}}$	Set of all variables
$val(S)$	Values of variables in S, in a particular state
$trm(S)$	Predicate that holds when S terminates
$abt(S)$	S never terminates
$FV(E)$	Set of free variables in E
z	A fresh variable

Table 3.1: General notations

We follow the convention in [1], [128], [68] of expressing $trm(S)$ in terms of the *negation* of an *abort* substitution, $abt(S)$. Given that $\neg$ denotes *negation*, and $[S]Q$ denotes the weakest precondition that the substitution S satisfies the predicate Q on termination, $abt(S)$ and $trm(S)$ are defined as:

$$abt(S) \Leftrightarrow \neg[S]Q \qquad \text{and}$$
$$trm(S) \Leftrightarrow \neg abt(S).$$

We adopt the general understanding in computer science that *expressions* refer to syntactic combinations of variables, constants and/or functions - combined by arithmetic, relational or other symbols. The standard meaning of *free variables* in an expression is a collection of variables (unbound by any quantifier) in the expression. Free variables are basically place-holders in an expression where values can be substituted to evaluate the expression. Hence, the value of an expression depends on the free variables within it. This explains the intuition behind the need to consider the free variables in expressions in order to determine variables whose values may *interfere* with the values of the variables updated by the expressions. Using the Backus-Naur Form (BNF) [4], [5], [140], we formally define an expression $E \in \mathbf{Exp}$ as follows:

$d_+ \in digit$; $\quad d_+ ::= \text{“1”}|\text{“2”}|\text{“3”}|\text{“4”}|\text{“5”}|\text{“6”}|\text{“7”}|\text{“8”}|\text{“9”}$;
$d \in digit$; $\quad d ::= \text{“0”} \mid d_+$;
$n \in number$; $\quad n ::= d \mid d_+\ d$;
$c \in character$; $\quad c ::= \text{“}A\text{”}|\text{“}a\text{”}|\text{“}B\text{”}|\text{“}b\text{”}|\text{“}C\text{”}|\text{“}c\text{”}|\text{“}D\text{”}|\text{“}d\text{”}|\text{“}E\text{”}|$
“e”|“F”|“f”|“G”|“g”|“H”|“h”|“I”|“i”|“J”|“j”|“K”|“k”|“L”|
“l”|“M”|“m”|“N”|“n”|“O”|“o”|“P”|“p”|“Q”|“q”|“R”|“r”|“S”|
“s”|“T”|“t”|“U”|“u”|“V”|“v”|“X”|“x”|“Y”|“y”|“Z”|“z”;
$w \in word$; $\quad w ::= c\ c \mid c\ n \mid w\ c \mid w\ n$;
$b \in binaryOP$; $\quad b ::= \text{“}-\text{”}|\text{“}+\text{”}|\text{“}=\text{”}|\text{“}*\text{”}|\text{“}/\text{”}|\text{“}\neq\text{”}|\text{“}<''\text{”}|\text{“}<=\text{”}|$
“$>$”|“$>=$”|“$\wedge$”|“$\vee$”|“$\Rightarrow$”|“$\cup$”|“$\cap$”|“$\in$”|“$\notin$”|“$\subset$”|“$\subseteq$”;

$E \in \mathbf{Exp}$;
$E ::= w \mid E\ b\ n \mid E\ b\ E$;

$$FV \in (\mathbf{Exp}) \rightarrow \mathcal{P}(\mathbf{Ide})$$

$$FV(E) = \begin{cases} \{w\}, & if\ E \equiv w \\ \{w\}, & if\ E \equiv w\ b\ n \\ \{w_1, w_2\}, & if\ E \equiv w_1\ b\ w_2 \\ FV(E_1) \cup FV(E_2), & if\ E \equiv E_1\ b\ E_2 \end{cases}$$

Other notations used in this chapter include $Q[E/x]$, which denotes the substitution of expression, E, for all occurrences of x in Q; n, which ranges over values of type NAT [1], which in B Method parlance stands for a set of natural numbers. ff denotes a function, while $\lhd\!\!+$ represents function update. Following convention, we write $:\in$ to denote assignment of a member of a set (written on the RHS) to a variable (written on the LHS), e.g., $n :\in N$ means a member of the set N is assigned to the variable n.

We distinguish between *abstract* and *concrete substitutions*[1] in our notation to capture the fact that certain substitutions allowed in abstract B machines are not allowed in implementations, and vice versa. For example, protected assignments (or preconditioned substitutions), PRE; nondeterministic substitutions like CHOICE, [], and ANY; SELECT, LET and multiple substitutions, $\|$, are not permitted in *implementations*, whereas sequential composition, ";", VAR and $WHILE$ substitutions are not allowed in abstract B machines [68], [1]. Following standard practice, we use x to range over *read-write* variables updated in a substitution, whereas we use z to range over *read-only* variables.

We now present the syntax of the GSL.

[1] abstract substitutions, ${}_aC$ - allowed only in machines;
concrete substitutions, ${}_\star C$ - allowed only in implementations;
a mix of some of both allowed in refinements.
Substitutions allowed in machines, refinements, and implementations, denoted C.

$$
\begin{array}{lll}
S & ::= & C \mid {}_{a}C \mid {}_{\star}C; \qquad \text{- - } \textit{(labelled substitutions)} \\
C & ::= & \textit{skip} \mid x := E \mid ff(x) := E \mid \\
 & & \quad \textbf{IF } b \textbf{ THEN } S \textbf{ END} \mid \\
 & & \quad \textbf{IF } b \textbf{ THEN } S_1 \textbf{ ELSE } S_2 \textbf{ END} \mid \\
 & & \quad \textbf{IF } b_1 \textbf{ THEN } S_1 \textbf{ ELSIF } b_2 \textbf{ THEN } S_2 \ldots \\
 & & \qquad \textbf{ELSE } S_n \textbf{ END}; \\
{}_{a}C & ::= & x_1 := E_1 \parallel x_2 := E_2 \mid \textbf{PRE } P \textbf{ THEN } {}_{a}C \textbf{ END} \\
 & & \quad \textbf{ANY } z \textbf{ WHERE } Q \textbf{ THEN } {}_{a}C \textbf{ END} \mid \\
 & & \quad \textbf{CHOICE } {}_{a}C_1 \textbf{ OR } {}_{a}C_2 \textbf{ END} \mid \\
 & & \quad \mid \textbf{SELECT } Q \textbf{ THEN } {}_{a}C_1 \textbf{ WHEN } R \textbf{ THEN } {}_{a}C_2 \ldots \\
 & & \qquad \textbf{ELSE } {}_{a}C_n \textbf{ END} \mid x :\in U \mid \\
 & & \quad \textbf{LET } x \textbf{ BE } x = E \textbf{ IN } {}_{a}C \textbf{ END} \mid C; \\
{}_{\star}C & ::= & \textbf{VAR } x \textbf{ IN } {}_{\star}C \textbf{ END} \mid C;
\end{array}
\tag{3.1}
$$

Having introduced our specialised syntax of GSL we now present the corresponding GSL semantics using Dijkstra's *weakest precondition (wp)* semantics. While Clark et al [36] used big-step semantics in their framework, we elect to use *wp* semantics in our work for the following reasons:

- Most existing work in the literature involving the B method define GSL semantics using *wp* semantics, so we reckon it fits more naturally with related literature to use *wp* semantics here.
- For the same reason, our information flow analysis framework will be more understandable and attractive to practitioners trained in the B Method.
- *wp* semantics is sufficiently robust and expressive for our definitions and proofs.

We present now the semantics of GSL in Table (3.2), followed with explanations of some of the main entries in the table.

Since *skip* is a substitution that does nothing, it follows that a predicate, P, holds after *skip* is called if and only if P holds before the call, thus,

Name (or AMN Statements)	wp Semantics
$skip$	$P \Leftrightarrow [skip]P$
$x \coloneqq E$	$P[E/x]$
$ff(x) \coloneqq E$	$[ff \coloneqq \{x \mapsto E\}]P \;\equiv\; P[ff <+ \{x \mapsto E\}/ff]$
$x_1 \coloneqq E_1 \parallel\; x_2 \coloneqq E_2$	$[x_1 \coloneqq E_1]\,[x_2 \coloneqq E_2]P$
SELECT Q **THEN** ${}_aC_1$ **WHEN** R **THEN** ${}_aC_2$ ... **ELSE** ${}_aC_n$ **END**	$[Q \Longrightarrow {}_aC_1\ []\ R \Longrightarrow {}_aC_2\ []...[]\ {}_aC_n]P$ $\equiv\; Q \Rightarrow [\ {}_aC_1]P \;\wedge\; R \Rightarrow [\ {}_aC_2]P \;\wedge ...$ $\wedge\ \neg(Q \vee R \vee ...) \;\Rightarrow\; {}_aC_n$
CHOICE ${}_aC_1$ **OR** ${}_aC_2$ **END**	$[\ {}_aC_1\ []\ {}_aC_2]P \;\equiv\; [\ {}_aC_1]P \;\wedge\; [\ {}_aC_2]P$
ANY z **WHERE** Q **THEN** ${}_aC$ **END**	$[@z \cdot\ (Q \Longrightarrow {}_aC)]P$ $\equiv\; \forall z \cdot (Q \Rightarrow [\ {}_aC]P)$ $\quad {}_{z \backslash Q}$
$x :\in U$	$[@z \cdot ((z \in U) \;\Longrightarrow\; (x \coloneqq z))]P \;\equiv$ $\forall z \cdot ((z \in U) \;\Rightarrow\; [x \coloneqq z]P)$ $\quad {}_{z \backslash pred(U)}$
PRE R **THEN** ${}_aC$ **END**	$[R \mid {}_aC]P \;\equiv\; R \wedge [\ {}_aC]P$
LET x **BE** $x = E$ **IN** ${}_aC$ **END**	$[@x \cdot ((x = E) \Longrightarrow {}_aC)]P \;\equiv$ $\forall x \cdot ((x = E) \;\Rightarrow\; [\ {}_aC]P)$ $\quad {}_{x \backslash E}$
IF b **THEN** S **END**	$[$**IF** b **THEN** S **ELSE** $skip$ **END**$]P$ $\equiv (b \;\Rightarrow\; [S]P) \;\wedge\; (\neg b \;\Rightarrow\; P)$
IF b **THEN** S_1 **ELSE** S_2 **END**	$[(b \Longrightarrow S_1)\ []\ (\neg b \Longrightarrow S_2)]P \;\equiv$ $(b \Rightarrow [S_1]P) \;\wedge\; (\neg b \Rightarrow [S_2]P)$
IF b_1 **THEN** S_1 **ELSIF** b_2 **THEN** S_2 ... **ELSE** S_n **END**	$[(b_1 \Longrightarrow S_1)\ []\ (b_2 \Longrightarrow S_2)\ []\ \cdot\cdot$ $(\neg(b_1 \vee\ b_2 \vee \dots) \Longrightarrow\ S_n)]P \;\equiv$ $b_1 \Rightarrow [S_1]P \;\wedge\; b_2 \Rightarrow [S_2]P\ \dots$ $(\neg(b_1 \vee\ b_2 \vee \dots) \Rightarrow [S_n]P)$
VAR x **IN** ${}_\star C$ **END**	$[@x \cdot\ {}_\star C]P$

Table 3.2: Semantics of GSL

$P \Leftrightarrow [skip]P$ correctly defines this meaning. We illustrate the meaning of function update, $ff(x) \coloneqq E$, with an example. Suppose we define a function $drivers$ mapping members of a set $DRIVERS$ to a set $TRAINS$, i.e., $drivers \in DRIVERS \rightarrow TRAINS$. Then, given that $tj \in DRIVERS$ and $T201 \in TRAINS$, we can update the fact that tj is the driver on train $T201$ by writing $driver(tj) \coloneqq T201$, meaning that all members of function $drivers$ remain unchanged, except for '$tj \mapsto ?$' (? denotes any member of $TRAINS$), which is now updated to $tj \mapsto T201$. Hence $[drivers \coloneqq \{tj \mapsto T201\}]P \equiv P[drivers \Leftarrow \{tj \mapsto T201\}/drivers]$.

The bounded choice substitution, ${}_aC_1 \; [] \; {}_aC_2$, allows the implementer the freedom to choose between a fixed number of alternative substitutions. Each alternative substitution must therefore satisfy the specified postcondition, P, hence $[\,{}_aC_1 \,[]\, {}_aC_2]P \equiv [\,{}_aC_1]P \wedge [\,{}_aC_2]P$. This conjunction over each alternative substitution is generalised to a universal quantifier, $\forall z \cdot Q \Rightarrow [\,{}_aC]P$, to define the semantics of unbounded choice substitution by Abrial [1]. Guarded substitutions have the meaning that the guard must hold for the substitution to be *feasible*, otherwise the substitution can establish anything, i.e., $[Q \Longrightarrow {}_aC_1]P \Leftrightarrow Q \Rightarrow [\,{}_aC_1]P$. The semantics of $SELECT$ describes bounded choice over guarded substitutions, hence for the substitution to be *feasible* all the guards must hold on some values. Preconditioned (or protected) substitution has the meaning that the precondition must hold for the substitution to successfully terminate, otherwise the substitution *cannot* establish anything, i.e., it *aborts*. $x_1 \coloneqq E_1 \parallel x_2 \coloneqq E_2$ describes multiple substitutions (unlike sequential substitutions), in no particular order, but with the requirement that updated variables are unique. The **IF** ... **THEN** substitutions are reducible to bounded choice over guarded substitutions.

Recall that to solve the problem of preservation of confidentiality properties through refinement we proposed a decoupling of the problem into:

- The problem of preservation of the refinement relation, and
- The problem of preservation of confidentiality properties of interest.

Our contribution deals with the problem of preservation of confidentiality properties using our information flow analysis framework, which is discussed in Section 3.4. We do not make any modification to the traditional notion of

refinement in the literature, thus we expect all B machines, refinements and implementations to satisfy the *machine* consistency and refinement conditions in the literature. For this reason, we do not discuss these conditions here, but refer the interested reader to Abrial [1]. In the following section, though, we discuss the axioms, and theorems we will employ later in proving the correctness of our information flow analysis framework.

3.3.2 Axioms and Theorems

In this section, we present the *normalised form* of generalised substitutions, introduced by Abrial [1], which forms the core of the axioms and theorems relating to the correctness of GSL semantics. We start with an extension of the notations given earlier in Section 3.3.1. Here, we write x and x' whereby x is a (list of) state variable(s) in the abstract machine under consideration. x' is a (list of) *fresh* variable(s) *distinct from* x (and not free in the predicate, P, used to define the precondition in Theorem 1 below). Note that x' holds the final value(s) of x after execution of S; P gives the states on which S is guaranteed to terminate, while Q relates the initial state x of S with the final state x', i.e., Q holds in *exactly* those final states x' reachable by S. Since Theorem 1 is a well-known result in the literature [1], [128], and the correctness of the theorem is well documented by Abrial in [1], we simply introduce and use it here.

Theorem 1. *[Normalised Form of Generalised Substitutions] - All generalised substitutions, S, updating some (list of) variable(s) x, can be expressed in the form*

$$S \;\widehat{=}\; P|@x' \cdot (Q \Longrightarrow x := x'), \qquad\qquad x' \backslash P$$

We now present axioms relating to the conditions required for a substitution to establish a predicate. The minimal of these conditions is that the substitution *terminates* [41], [1]. As noted earlier, $trm(S)$ is defined *indirectly* in terms of its negation, an aborting substitution, $abt(S)$, which is a substitution that cannot establish any predicate (Table 3.1). Even the most obvious of predicates, $x = x$, cannot be established by $abt(S)$. Formally,

$$abt(S) \;\Leftrightarrow\; \neg[S]R, \qquad \text{for any predicate } R$$

$trm(S)$ Shape	Meaning
$trm(skip)$	$Noop$, terminates
$trm(x := E)$	variable update, "
$trm(ff(x) := E)$	function update, "
$trm(x :\in U)$	set choice, "
$trm(P \mid {}_aC)$	$\Leftrightarrow \quad P \wedge trm({}_aC)$
$trm(Q \Longrightarrow {}_aC)$	$\Leftrightarrow \quad Q \Rightarrow trm({}_aC)$
$trm({}_aC_1 [] {}_aC_2)$	$\Leftrightarrow \quad trm({}_aC_1) \wedge trm({}_aC_2)$
$trm(@z \cdot (Q \Longrightarrow {}_aC))$	$\Leftrightarrow \quad \forall z \cdot (Q \Rightarrow trm({}_aC))$
$trm(@x \cdot (x = E \Longrightarrow {}_aC))$	$\Leftrightarrow \quad \forall x \cdot ((x = E) \Rightarrow trm({}_aC))$
$trm(@x \cdot {}_\star C)$	$\Leftrightarrow \quad \forall x \cdot trm({}_\star C)$
$trm(P \mid @x' \cdot (Q \Longrightarrow x := x'))$	$\Leftrightarrow \quad P$

Table 3.3: Termination Property on shape of GSL

i.e., $abt(S) \Leftrightarrow \neg[S](x = x)$,
hence, $trm(S) \Leftrightarrow \neg abt(S)$
i.e., $trm(S) \Leftrightarrow [S]R$, for some predicate R
thus, $trm(S) \Leftrightarrow [S](x = x)$

Consequently, we summarise in Table 3.3 the shape of $trm(S)$ with respect to the semantics of GSL presented earlier in Table 3.2. Notice that the preconditioned substitution, $P \mid {}_aC$, requires that P holds, otherwise the substitution *aborts*, hence the conjunction $P \wedge trm({}_aC)$ is the termination condition. For guarded substitution, $Q \Longrightarrow {}_aC$, though, the fact that Q (the predicate that relates the initial state of the substitution with the final state) holds *implies* that the guarded substitution is not only feasible, but it also terminates, otherwise there can be no final state. Hence $Q \Rightarrow trm({}_aC)$.

Another important axiom introduced by [1] is the representation of GSL substitutions in terms of *before-after predicates*. This semantic notion along with $trm(S)$ was then used by the author to show the identity of generalised substitutions. This throws more light on the meaning of the normalised form of GSL presented earlier in Theorem 1. Given that x is a variable in the machine where the substitution S is defined, and the before-after predicate depends on x, we follow the convention in [1] and write $prd_x(S)$ to denote

the predicate which relates the value of x before S is executed to its after-value, denoted x', which is assumed to be a fresh variable. The before-after predicate notion is captured by the formula given below, which literally reads: *it is not the case that the substitution S establish that x and x'* are not equal after execution.

$$prd_x(S) \;\Leftrightarrow\; \neg[S](x' \neq x) \tag{3.2}$$

Given, for example, that $S \triangleq x := E$, then we will expect the substitution to terminate in a state satisfying the predicate $x' = E$. But if $S \triangleq y := E$, then we expect $prd_{x,y}(S)$ to hold *if and only if* $x' = x$ and $y' = E$, since x is not updated in the substitution. We present the proofs of before-after predicates with respect to the semantics of GSL as follows:

Skip:

$$\begin{aligned} &skip = x := x \\ &prd_x(skip) \;\Leftrightarrow\; \neg[x := x](x' \neq x) \\ &\qquad\qquad \Leftrightarrow\; \neg([x := x](x' \neq x)) \\ &\qquad\qquad \Leftrightarrow\; \neg(x' \neq x) \\ &\qquad\qquad \Leftrightarrow\; (x' = x) \end{aligned}$$

Simple Substitution (x:=E):

$$\begin{aligned} prd_x(x := E) \;&\Leftrightarrow\; \neg[x := E](x' \neq x) \\ &\Leftrightarrow\; \neg([x := E](x' \neq x)) \\ &\Leftrightarrow\; \neg(x' \neq E) \\ &\Leftrightarrow\; (x' = E) \end{aligned}$$

Simple Substitution (x:=E) - Multiple variables in Machine:

$$\begin{aligned} prd_{x,y}(x := E) \;&\Leftrightarrow\; \neg[x := E](x', y' \neq x, y) \\ &\Leftrightarrow\; \neg([x := E](x', y' \neq x, y)) \\ &\Leftrightarrow\; \neg(x', y' \neq E, y) \\ &\Leftrightarrow\; (x', y' = E, y) \\ &\Leftrightarrow\; (x' = E) \wedge (y' = y) \end{aligned}$$

Since only x is updated in S, we correctly have the before- and after-values of y equal, i.e., $y = y'$.

Multiple Substitution $(x := E_1 \parallel y := E_2)$ - Multiple variables updated:

$$\begin{aligned} prd_{x,y}(x := E_1 \parallel y := E_2) &\Leftrightarrow \neg[x, y := E_1, E_2](x', y' \neq x, y) \\ &\Leftrightarrow \neg([x, y := E_1, E_2](x', y' \neq x, y)) \\ &\Leftrightarrow \neg(x', y' \neq E_1, E_2) \\ &\Leftrightarrow (x', y' = E_1, E_2) \\ &\Leftrightarrow (x' = E_1) \wedge (y' = E_2) \end{aligned}$$

From the foregoing, it is a simple thing to extend the proof by induction, to other GSL constructs.

We now discuss the notion of *variable state comprehension predicate*, which requires that whenever x is assigned any value such that a postcondition predicate, P, is satisfied, it must be the case that there is a relationship between the before state and the after state of the variable(s) referenced in the substitution that satisfies P. Given that the before state of the variable x in comprehension is denoted x_0, and the after state is x'. Let P be a predicate. Then the variable state comprehension predicate is formally written as:

$$@x' \cdot [x_0, x := x, x']P \implies x := x' \tag{3.3}$$

This predicate statement, abbreviated $x : P$, basically states that x is updated to its after-value thereby moving the system from its before state, x_0, to the after state, x', which subsequently becomes the next before state.

We present in Table 3.4 the shape of $prd_x(S)$ for the GSL semantics defined in Table 3.2. Notice here that the definition of $prd_x(\ P \mid {}_aC)$ for preconditioned substitutions indicates that whenever ${}_aC$ is feasible, i.e., $prd_x(\ {}_aC)$ holds, then it must be the case that P holds too (since $P \Rightarrow prd_x(\ {}_aC)$). For guarded substitutions, $Q \implies {}_aC$, though, since Q relates the before-value to the after-value, Q must always hold for ${}_aC$ to be feasible, i.e., $Q \wedge prd_x(\ {}_aC)$.

Another important theorem proved by Abrial in [1], which we will employ in this thesis is stated below (Theorem 2).

Theorem 2. *[Predicate Normalised Form of Generalised Substitutions] - All generalised substitutions, S, can be presented in terms of the termination and*

$prd_x(S)$ Shape	Meaning	
$prd_x(skip)$	$\Leftrightarrow$	$x' = x$
$prd_x(x := E)$	$\Leftrightarrow$	$x' = E$
$prd_{x,y}(x := E)$	$\Leftrightarrow$	$x', y' = E, y$
$prd_x(ff(x) := E)$	$\Leftrightarrow$	$ff' = ff \nleftarrow \{x \mapsto E\}$
$prd_x(x :\in U)$	$\Leftrightarrow$	$x' \in U$
$prd_x(P \mid {}_aC)$	$\Leftrightarrow$	$P \Rightarrow prd_x({}_aC)$
$prd_x(Q \Longrightarrow {}_aC)$	$\Leftrightarrow$	$Q \wedge prd_x({}_aC)$
$prd_x(x = E \Longrightarrow {}_aC)$	$\Leftrightarrow$	$(x = E) \wedge prd_x({}_aC)$
$prd_x({}_aC_1 [] {}_aC_2)$	$\Leftrightarrow$	$prd_x({}_aC_1) \vee prd_x({}_aC_2)$
$prd_x(@z \cdot (Q \Longrightarrow {}_aC))$	$\Leftrightarrow$	$\exists z \cdot (Q \wedge prd_x({}_aC))$
$prd_x(@z \cdot (z = E \Longrightarrow {}_aC))$	$\Leftrightarrow$	$\exists z \cdot ((z = E) \wedge prd_x({}_aC))$
$prd_x(@z \cdot {}_\star C)$	$\Leftrightarrow$	$\exists z \cdot prd_x({}_\star C)$
$prd_x(P \mid @x' \cdot (Q \Longrightarrow x := x'))$	$\Leftrightarrow$	$P \Rightarrow Q$
$prd_x(x : P)$	$\Leftrightarrow$	$[x_0, x := x, x']P$

Table 3.4: Before-After Predicate on shape of GSL

before-after predicate as

$$S \;=\; trm(S) \mid @x' \cdot (prd_x(S) \Longrightarrow x := x')$$

We supply the proof of Theorem 2 given in [1], which we simplify and annotate here, since we will be employing the theorem later in Section 3.4 to prove the correctness of our information flow analysis framework with respect to the semantics presented in Tables 3.3, and 3.4.

To prove Theorem 2, it is sufficient to show that any postcondition established by the LHS is also established by the RHS [1], i.e., given a postcondition R, $[S]R$ holds if and only if $[trm(S) \mid @x'.(prd_x(S) \Longrightarrow x := x')]R$ holds.

Proof:

$LHS = [S]R$

$\Leftrightarrow\ [P \mid @x' \cdot (Q \Longrightarrow x := x')]R$ - (writing S in normal form)

$\Leftrightarrow\ P \wedge \forall x' \cdot (Q \Rightarrow [x := x']R), \qquad x' \backslash P$ - (PRE in Table 3.2)

$\Leftrightarrow\ P \wedge \forall x' \cdot ((P \Rightarrow Q) \Rightarrow [x := x']R)$ - (distributive law)

(recall from Table 3.4, $prd_x(P \mid @x' \cdot (Q \Longrightarrow x := x')) = P \Rightarrow Q$)

$$\Leftrightarrow\ P \wedge \forall x' \cdot (prd_x(P|@x' \cdot (Q \Longrightarrow x := x')) \Rightarrow [x := x']R)$$
$$\Leftrightarrow\ P \wedge TRUE \mid \forall x' \cdot (prd_x(S) \Rightarrow [x := x']R) \quad \text{- (contracting)}$$
$$\Leftrightarrow\ [trm(S)|@x' \cdot (prd_x(S) \Longrightarrow x := x')]R = RHS \ \text{-(Table 3.3)}$$

As noted by Abrial in [1], the consequence of Theorem 2 is that both $trm(S)$ and $prd_x(S)$ characterise completely the generalised substitution S defined on x. The author, [1], also noted that this corresponds to saying that $trm(S)$ and $trm(S) \Rightarrow prd_x(S)$ characterise the generalised substitution S. We present in Section 3.4 our flow analysis framework for GSL.

3.4 Information Flow Analysis of Generalised Substitutions

In this section, we extend the flow logic approach to information flow analysis developed by Clark et al [36], [118], [7] to the flow analysis of GSL. Recall that we introduced a special metavariable $\bullet$ in Table 3.1. We now define $\widehat{\mathbf{Ide}} \supseteq \mathbf{Ide} \cup \{\bullet\}$. In addition to the notations introduced in Table 3.1, we define the three key functions required for our analysis, respectively named 'Assign', 'Global', and 'Dep' as follows:

$\widehat{X} \in \mathit{Assign} = \mathbf{Lab} \to \mathcal{P}(\mathbf{Ide})$
$\widehat{G} \in \mathit{Global} = \mathbf{Lab} \to \mathcal{P}(\widehat{\mathbf{Ide}})$
$\widehat{D} \in \mathit{Dep} = \mathbf{Lab} \to \mathcal{P}(\widehat{\mathbf{Ide}} \times \widehat{\mathbf{Ide}})$

For any given label, ℓ:
$\widehat{X}(\ell)$ returns the superset of all variables that may be assigned to. This corresponds to the notion of *write frame* in Bicarregui's work [23];
$\widehat{G}(\ell)$ returns the superset of variables whose values may affect termination of the statement with label ℓ;
Note: A pair (x, y) is a dependency in a substitution S, if a change in the value of y in the state before execution of S can result in different values for x after termination of S.
$\widehat{D}(\ell)$ returns the superset of pairs of variables which constitute dependencies for the statement with label ℓ.
Id denotes an identity relation on $\widehat{\mathbf{Ide}}$, e.g., given that the variable x depends only on itself (i.e. $\widehat{D} = (x, x)$), we write $\widehat{D} \supseteq Id$.
We write $(\widehat{G}, \widehat{D}) \models S$ to denote the notion that $(\widehat{G}, \widehat{D})$ is an *acceptable*

information flow analysis of S, and inductively define $\widehat{X}$ on the semantics of GSL as follows.

$$\widehat{X}(\ell) = \begin{cases} \varnothing, & C^\ell \equiv skip \\ \{x\}, & C^\ell \equiv x := E \\ \{ff\}, & C^\ell \equiv ff(x) := E \\ \widehat{X}(\ell_1) \cup \widehat{X}(\ell_2), & {}_aC^\ell \equiv \ (x_1 := E_1)^{\ell_1} \parallel \ (x_2 := E_2)^{\ell_2} \\ \widehat{X}(\ell_1), & {}_aC^\ell \equiv Q \Longrightarrow \ {}_aC_1^{\ell_1} \\ \widehat{X}(\ell_1) \cup \widehat{X}(\ell_2), & {}_aC^\ell \equiv \ {}_aC_1^{\ell_1} \ [] \ {}_aC_2^{\ell_2} \\ \widehat{X}(\ell_1), & {}_aC^\ell \equiv P \mid {}_aC_1^{\ell_1} \\ \widehat{X}(\ell_1), & {}_aC^\ell \equiv @z \cdot (Q \Longrightarrow \ {}_aC_1^{\ell_1}) \\ \widehat{X}(\ell_1), & {}_aC^\ell \equiv @z \cdot ((z \in U) \Longrightarrow \ {}_aC_1^{\ell_1}) \\ \widehat{X}(\ell_1), & {}_aC^\ell \equiv @x \cdot ((x = E) \Longrightarrow \ {}_aC_1^{\ell_1}) \\ \bigcup_{i=1}^{n} \widehat{X}(\ell_i), & {}_aC^\ell \equiv Q \Longrightarrow \ {}_aC_1^{\ell_1} \ [] \ R \ \Longrightarrow \ {}_aC_2^{\ell_2} \ []...[] \ {}_aC_n^{\ell_n} \\ \bigcup_{i=1}^{n} \widehat{X}(\ell_i), & S \equiv b_1 \ \Longrightarrow \ S_1 \ [] \ b_2 \ \Longrightarrow \ S_2 \ []...[] \ \Longrightarrow \ S_n \\ \widehat{X}(\ell_1), & {}_\star C^\ell \equiv @ \ x \cdot \ {}_\star C_1^{\ell_1} \end{cases}$$

We now extend the information flow analysis presented in ([36], [118], [7]) to the substitutions defined in our specialised GSL. As in [36], we employ the special character $\bullet$ to denote the notion that there is the possibility of nontermination, e.g., when a precondition fails in a protected substitution.

$$(\widehat{G}, \widehat{D}) \ \vDash \ skip^\ell \ \Leftrightarrow \ \widehat{D}(\ell) \supseteq Id$$
$$(\widehat{G}, \widehat{D}) \ \vDash \ (x := E)^\ell \quad \Leftrightarrow \widehat{D}(\ell) \supseteq Id[x \mapsto FV(E)]$$

$$(\widehat{G}, \widehat{D}) \ \vDash \ (ff(x) := E)^\ell \quad \Leftrightarrow \widehat{D}(\ell) \supseteq Id[ff(x) \mapsto FV(E)]$$

$$\begin{aligned} (\widehat{G}, \widehat{D}) \ \vDash \ & ((x_1 := E_1)^{\ell_1} \parallel \ (x_2 := E_2)^{\ell_2})^\ell \\ & \Leftrightarrow (\widehat{G}, \widehat{D}) \ \vDash \ (x_1 := E_1)^{\ell_1} \ \wedge \ (\widehat{G}, \widehat{D}) \ \vDash \ (x_2 := E_2)^{\ell_2} \ \wedge \\ & \quad \widehat{D}(\ell) \supseteq Id[x_1 \mapsto FV(E_1)] \ \cup \ Id[x_2 \mapsto FV(E_2)] \end{aligned}$$

$$\begin{aligned} (\widehat{G}, \widehat{D}) \ \vDash \ & (Q \Longrightarrow C_1^{\ell_1})^\ell \\ & \Leftrightarrow (\widehat{G}, \widehat{D}) \ \vDash \ C_1^{\ell_1} \ \wedge \\ & \quad \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \ \wedge \ \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(Q)) \end{aligned}$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (P|C_1^{\ell_1})^{\ell} \\
&\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; C_1^{\ell_1} \;\wedge\; (\bullet \in \widehat{G}(\ell) \Rightarrow \widehat{G}(\ell) \supseteq FV(P)) \;\wedge \\
&\quad\; \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \;\wedge\; \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(P))
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (@z \cdot (z \in U) \Longrightarrow C^{\ell_1})^{\ell} \\
&\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; C^{\ell_1} \;\wedge \\
&\quad\; \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \;\wedge\; \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(pred(U)))
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (@z \cdot (z = E) \Longrightarrow C^{\ell_1})^{\ell} \;\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; C^{\ell_1} \;\wedge \\
&\quad\; \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \;\wedge\; \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(E))
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (\ {}_aC_1^{\ell_1}[]\ {}_aC_2^{\ell_2})^{\ell} \\
&\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; {}_aC_1^{\ell_1} \;\wedge\; (\widehat{G},\widehat{D}) \;\vDash\; {}_aC_2^{\ell_2} \;\wedge \\
&\quad\; \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \cup \widehat{G}(\ell_2) \;\wedge\; \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup \widehat{D}(\ell_2)
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (\textbf{SELECT}\ Q\ \textbf{THEN}\ {}_aC_1^{\ell_1}\ \textbf{WHEN}\ R\ \textbf{THEN}\ {}_aC_2^{\ell_2} \\
&\qquad \ldots\ \textbf{ELSE}\ {}_aC_n^{\ell_n}\ \textbf{END})^{\ell} \\
&\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; {}_aC_1^{\ell_1} \wedge (\widehat{G},\widehat{D}) \;\vDash\; {}_aC_2^{\ell_2} \wedge \ldots \wedge (\widehat{G},\widehat{D}) \;\vDash\; {}_aC_n^{\ell_n} \;\wedge \\
&\quad\; \widehat{G}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{G}(\ell_i) \;\wedge\; \widehat{D}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{D}(\ell_i) \;\cup\; (\widehat{X}(\ell) \times (\bigcup_{i=1}^{n-1} FV(b_i)))
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (@z \cdot Q \Longrightarrow C_1^{\ell_1})^{\ell} \\
&\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; C_1^{\ell_1} \;\wedge\; \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \;\wedge \\
&\quad\; \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times (FV(Q) \cup \{z\}))
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (\textbf{IF}\ b_1\ \textbf{THEN}\ C_1^{\ell_1}\ \textbf{ELSIF}\ b_2\ \textbf{THEN}\ C_2^{\ell_2} \ldots \textbf{ELSE}\ C_n^{\ell_n}\ \textbf{END})^{\ell} \\
&\Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; C_1^{\ell_1} \wedge (\widehat{G},\widehat{D}) \;\vDash\; C_2^{\ell_2} \;\wedge \ldots \wedge (\widehat{G},\widehat{D}) \;\vDash\; C_n^{\ell_n} \;\wedge \\
&\quad\; \widehat{G}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{G}(\ell_i) \;\wedge\; \widehat{D}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{D}(\ell_i) \;\cup\; (\widehat{X}(\ell) \times (\bigcup_{i=1}^{n-1} FV(b_i))) \\
&\quad\; \wedge\; (\bullet \in \widehat{G}(\ell) \Rightarrow \widehat{G}(\ell) \supset (\bigcup_{i=1}^{n-1} FV(b_i))
\end{aligned}
$$

$$
\begin{aligned}
(\widehat{G},\widehat{D}) \;&\vDash\; (\textbf{VAR}\ x\ \textbf{IN}\ {}_\star C_1^{\ell_1})^{\ell} \\
&\quad\; \Leftrightarrow (\widehat{G},\widehat{D}) \;\vDash\; C_1^{\ell_1} \wedge\; \widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \;\wedge\; \widehat{G}(\ell) \supseteq \widehat{G}(\ell_1)
\end{aligned}
$$

Table 3.4b: Information Flow Analysis for GSL.

Notice from Table 3.4b that the information flow analysis for *skip* is acceptable *only if* the dependencies in the substitution after execution remains the same as before execution. Hence, $\widehat{D}(\ell) \supseteq Id$. In $x := E$, only x is updated, hence $\widehat{X} \supseteq \{x\}$, and throughout this thesis, we abbreviate $\{x\} \rightarrow FV(E)$ to $x \mapsto FV(E)$, meaning that x maps *pointwise* to each variable in $FV(E)$. Information flows from $FV(E)$ into x, i.e., given that $\stackrel{def}{=}$ denotes 'is defined as', $\widehat{D}(\ell) \supseteq Id[x \mapsto FV(E)]$, where $x \mapsto FV(E) \stackrel{def}{=} \{(x, y) \mid (\exists y \in FV(E))\ x \mapsto y\}$. The analysis of multiple substitutions yield the union of the dependencies of each substitution. Ditto the set of variables assigned to.

The flow analysis of guarded substitution, $(Q \Longrightarrow {}_aC_1^{\ell_1})^{\ell}$, demands that the free variables of the guard, Q, flow into the variable(s) assigned to in the body of the substitution, ${}_aC_1^{\ell_1}$, since the guard must hold for the substitution to be *feasible*. Hence, $\widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(Q))$. Similarly, for **SELECT** substitution, information flows from the free variables of the conditional(s) into the variable(s) updated in the body of the substitution. The same is also true for the **ANY** GSL substitution.

The analyses of the GSL substitutions discussed thus far do not contain any global flow, reason being that global flows are only possible in preconditioned substitutions and loops. In the remaining part of this section, we discuss the substitutions wherein global flows may occur. Clark et al in [36] showed that a program may not terminate if a *while* loop is encountered in the body of the program. Since Abrial's definition of GSL in [1] does not include loops and sequential compositions (not permissible in specifications), we do not include these substitutions in our specialised syntax of GSL. Hence, readers interested in how our analysis framework deals with these substitutions are referred to [36].

We also note that Clark et al [36] introduced and proved the flow analysis for *standard* **IF** statements, but provided no definition for the B GSL **IF-ELSIF** statement. Hence, we extend [36]'s language to include the B GSL **IF-ELSIF** statement. We note that a loop may be encountered in the body of the **IF-ELSIF** substitution (if used in implementations), in which case the '**IF-ELSIF** conditions' may affect the termination of

the substitution and all subsequent substitutions, if any. Hence, we add $\bullet \in \widehat{G}(\ell) \Rightarrow \widehat{G}(\ell) \supset (FV(b_1) \cup FV(b_2) \cup ...)$ to our analysis for *global* flows. In Section 3.3 we pointed out that a preconditioned (or protected) substitution fails to terminate (i.e., *aborts*) if the precondition does not hold prior to execution of the substitution, in which case the program in which the substitution is defined may simply *crash*. Hence, in Table 3.4b, we define the notion that a preconditioned substitution (and other subsequent substitutions) may not terminate by adding $\bullet \in \widehat{G}(\ell) \Rightarrow \widehat{G}(\ell) \supset FV(P)$ to the analysis, since the free variables of P determines whether the substitution terminates or not.

Example: Information Flow Analysis of PRE Substitution.
We provide here an information flow analysis of the *update_Cw* operation defined earlier in Table 2.7. A labelled version of this operation is re-presented below for ease of reference:

$$
\begin{array}{l}
mm \longleftarrow update\_Cw(regNum,\ mark) \ = \\
\quad (\textbf{PRE} \\
\qquad regNum \in \textbf{NAT1} \ \wedge \ regNum <= maxInteger \ \wedge \\
\qquad mark \in \textbf{NAT} \ \wedge \ mark <= maxMark \ \wedge \\
\qquad \textbf{mark = exam\_H\_r} \\
\quad \textbf{THEN} \\
\qquad (\textbf{cw\_L\_r} := \textbf{mark})^{\ell_1} \ ; \\
\qquad (\textbf{cwMarkPair\_r(regNum)} := \textbf{mark})^{\ell_2} \ ; \\
\qquad (\textbf{mm} := \textbf{mark})^{\ell_3} \\
\quad \textbf{END})^{\ell_0}
\end{array}
$$

We illustrate in Table 3.5 the collection of the information flow constraints for each of the labelled substitutions in terms of our $\widehat{X}, \widehat{G}, \widehat{D}$ abstraction. Now, suppose the variable *exam_H_r* is a high security variable whereas all other variables in the substitution *update_Cw* are low security variables, then the fact that **mark = exam_H_r** is used in the precondition means that an adversary with access to the program text can *implicitly* deduce the "*secret*" value of *exam_H_r* by simply observing the output on the low security variable *mm* on animation of the machine containing the substitution. Although, at this juncture, we are yet to introduce the *security conditions*

used in our information flow analysis, it is clear from this simple example that both explicit and implicit flows of information can be detected by our analysis framework.

Information Flow Analysis: Constraints Computation			
		ConstraintsCollectors	
Labels	$\widehat{X}$	$\widehat{G}$	$\widehat{D}$
ℓ_0	$\{cw\_L\_r,\ mm,$ $cwMarkPair\_r\}$	$\{regNum,$ $maxInteger,$ $mark,$ $maxMark,$ $exam\_H\_r\}$	$\{cw\_L\_r \mapsto regNum,$ $cw\_L\_r \mapsto maxInteger,$ $cw\_L\_r \mapsto mark,\ cw\_L\_r \mapsto maxMark,$ $cw\_L\_r \mapsto exam\_H\_r,\ mm \mapsto exam\_H\_r,$ $mm \mapsto regNum,\ mm \mapsto maxINteger,$ $mm \mapsto mark,\ mm \mapsto maxMark,$ $cwMarkPair\_r \mapsto regNum,$ $cwMarkPair\_r \mapsto maxInteger,$ $cwMarkPair\_r \mapsto mark,$ $cwMarkPair\_r \mapsto maxMark,$ $cwMarkPair\_r \mapsto exam\_H\_r\}$
ℓ_1	$\{cw\_L\_r\}$	$\varnothing$	$\{cw\_L\_r \mapsto mark\}$
ℓ_2	$\{cwMarkPair\_r\}$	$\varnothing$	$\{cwMarkPair\_r \mapsto mark\}$
ℓ_3	$\{mm\}$	$\varnothing$	$\{mm \mapsto mark\}$

Table 3.5: Information Flow Constraints Computation Example

3.4.1 Abstraction and Correctness Analysis of GSL

To analyse information flow security in GSL substitutions, we abstract GSL semantics as an over-approximation over the sets $\widehat{X}, \widehat{G}, \widehat{D}$ defined in Table 3.4b. Hence we formalise our notion of *secure flow analysis* as a mapping of the abstraction to a two-valued (boolean) range as shown in Formula 3.4,

for any substitution, S.

$$secure(S) \triangleq (\widehat{X}, \widehat{G}, \widehat{D}) \longrightarrow \{true, false\} \tag{3.4}$$

We illustrate our abstraction framework with the GSL construct, S, below:

(IF $p = q$ **THEN**
$(x := y + w)^{\ell_2}$
ELSE
$(z := 0)^{\ell_1}$
END)$^{\ell_0}$

Using Table 3.4b, we over-approximate the semantics of S in terms of $\widehat{X}$, $\widehat{G}$, and $\widehat{D}$ as follows:

$$\widehat{X}(\ell_0) = \widehat{X}(\ell_1) \cup \widehat{X}(\ell_2) = \{x, z\}$$

$$\begin{aligned}\widehat{D}(\ell_0) &= \widehat{D}(\ell_1) \cup \widehat{D}(\ell_2) \cup (\widehat{X}(\ell_0) \times FV(p = q)) \\ &= \{(z, \varnothing)\} \cup \{(x, y), (x, w)\} \cup \{(z, p), (z, q), (x, p), (x, q)\}. \\ &= \{(x, y), (x, w), (x, p), (x, q), (z, p), (z, q)\}, \text{ and}\end{aligned}$$

$$\widehat{G}(\ell_0) = \varnothing$$

Using the foregoing approximation, we conclude that our $(\widehat{X}, \widehat{G}, \widehat{D})$ abstraction of S is secure (or evaluates to $true$) if all variables, v_2, on which the variables, v_1, in $\widehat{X}$ depend in all possible runs of the program have security classification(s) lower than or equal to the security classification of v_1.

Suppose **Val** denotes a set of integer values, and σ denotes stores (or states), where $\sigma \in \mathbf{Ide} \to \mathbf{Val}$, i.e.:

$$\sigma = \{(x \mapsto \sigma(x)) \mid x \in \mathbf{Ide}\}$$

Given that Σ denotes a set of stores, i.e., $\Sigma \in \mathcal{P}(\mathbf{Ide} \to \mathbf{Val})$, it follows that for all $\sigma_1, \sigma_2, ..., \sigma_n \in \Sigma$,

$$\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$$

Given a set of variables X, Clark et al [36] defined *equivalence on stores* with respect to X, denoted $\approx^X$, to mean that two stores agree on all $x \in X$, i.e., writing $\sigma(x)$ to denote the value of variable x in state σ:

$$\sigma_1 \approx^X \sigma_2 \;\Leftrightarrow\; \forall x \in X \cdot \sigma_1(x) = \sigma_2(x) \tag{3.5}$$

We extend this notion of equivalence on stores to *equivalence on sets of stores*. We say that two sets of stores $\Sigma_1,\ \Sigma_2$ are equivalent with respect to a set of variables X, written $\Sigma_1 \approx^X \Sigma_2$ if and only if:

$$\forall \sigma_1 \in \Sigma_1,\ \exists \sigma_2 \in \Sigma_2 \cdot \sigma_1 \approx^X \sigma_2 \text{ and } \forall \sigma_2 \in \Sigma_2,\ \exists \sigma_1 \in \Sigma_1 \cdot \sigma_1 \approx^X \sigma_2 \tag{3.6}$$

It is easy to see that our extended definition of equivalence on sets of stores subsumes the definition by Clark et al [36] *on stores*, for the reason that application of the former to *singleton* sets of stores *transparently* corresponds to an application of the the latter to stores. That is, given that $\Sigma_1 = \{\sigma_1\}$ and $\Sigma_2 = \{\sigma_2\}$, then $\Sigma_1 \approx^X \Sigma_2$ is equivalent to $\sigma_1 \approx^X \sigma_2$.

Note: Clark et al in [36] provided proofs relating to three aspects of correctness with respect to their imperative (*While*) language based analysis framework, namely that:

- Their analysis is well-defined;
- Their analysis results are a proper abstraction of the semantics; and
- Every program has an acceptable information flow analysis and the constraints have solutions.

We note that it is not necessary to show that our analysis in this thesis is well-defined, since it is an extension of the structure already proved to be well-defined by Clark et al [36]. However, we prove the remaining two aspects of correctness itemised above with respect to the application of our information flow analysis framework to GSL semantics. The authors [36] showed, by structural induction on the abstract syntax tree, that their analysis results are a proper abstraction of the semantics of their core imperative language. In this section, we also use structural induction to develop the

correctness proofs on the predicate normalised form presented in Theorem 2 since all generalised substitutions have been shown to be expressible in this form [1].

NOTE: We first translate the predicate normalised form of GSL to a form amenable to our proof method.

Recall from Section 3.3, Theorem 2 that, for any substitution, C^ℓ:

$C^\ell \;=\; trm(C^\ell) \mid @x \cdot (prd_x(C^\ell) \Longrightarrow x := x')$

Abrial [1] showed that this corresponds to:

$$trm(C^\ell) \;\wedge\; trm(C^\ell) \Rightarrow prd_x(C^\ell) \tag{3.7}$$

But we know that:

$trm(C^\ell) \Rightarrow prd_x(C^\ell) \;\equiv\; \neg trm(C^\ell) \vee prd_x(C^\ell)$ - material implication

Hence Formula 3.7 becomes:

$trm(C^\ell) \;\wedge\; (\neg trm(C^\ell) \vee prd_x(C^\ell))$

$(trm(C^\ell) \wedge \neg trm(C^\ell)) \;\vee\; (trm(C^\ell) \wedge prd_x(C^\ell))$ - distributive law

$FALSE \;\vee\; (trm(C^\ell) \wedge prd_x(C^\ell))$ - simplification

$trm(C^\ell) \wedge prd_x(C^\ell)$

We therefore use $trm(C^\ell) \wedge prd_x(C^\ell)$ to prove Theorems 3 and 4 that are intended to show that our information flow analysis framework is a correct and proper abstraction of the semantics of our abridged GSL. In both cases, we assume $trm(C^\ell)$ holds, so we are only left to show that $prd_x(C^\ell)$ also holds in their respective contexts.

We use the conventional 'primed' notation on sets of stores, Σ_1', to denote the set of variable-value mappings corresponding to the *after-states* of the system after multiple animations (or runs) of a substitution.

Theorem 3 (Assignment Freedom (GSL))**.** *Whenever a substitution terminates, the after-values of the set of variables not assigned to equals the before-values of the same set of variables. Suppose* $(\widehat{G}, \widehat{D}) \vDash C^\ell$ *and let* $X^c = \{y \in \mathbf{Ide} \mid y \notin \widehat{X}(\ell)\}$*. Given that* $D_x \triangleq \widehat{D}(\ell)(x)$ *denotes, for all* $x \in \widehat{X}(\ell)$*, the set of variables on which* x *depends at* ℓ *(i.e., read frame by Bicarregui's definition in [23]), then:*

For any sets of *before* and *after* states Σ, Σ' respectively of a machine C^ℓ

where x is a (list of) variable(s) updated,

y is a (list of) variable(s) independent of x (i.e., $y \in X^c$), and

$\lhd$ denotes domain restriction,

we need to show that, *with respect to* x:

(1). if $prd_{x,y}(C^\ell)$ then $\Sigma \approx^{X^c} \Sigma'$

(2). $X^c \lhd \widehat{D}(\ell) = Id$

PROOF: Assume C^ℓ ranges over the GSL syntax tree, we employ structural induction to prove Theorem 3 as follows:

Part 1:

Case $C^\ell \triangleq$ *skip*: It trivially follows that both parts hold.

Case $C^\ell \triangleq x := E$, then:

$$\begin{aligned} prd_{x,y}(x := E) &\Leftrightarrow \neg[x := E](x', y' \neq x, y) && \text{- from Formula 3.2} \\ &\Leftrightarrow \neg([x := E](x', y' \neq x, y)) \\ &\Leftrightarrow \neg(x', y' \neq E, y) \\ &\Leftrightarrow (x', y' = E, y) \\ &\Leftrightarrow (x' = E) \wedge (y' = y) \end{aligned}$$

Hence, $\Sigma \approx^{X^c} \Sigma'$ since y ranges over X^c and $y' = y$.

Part 2: It follows from part 1 that the after-value of y, i.e., y', depends only on the before-value of y for all $y \in X^c$. Hence, $X^c \lhd \widehat{D}(\ell) = Id$ holds.

Note: Since x is either a variable or a list of variables in $\widehat{X}(\ell)$, the proof given above also covers the case where $C^\ell \triangleq x_1 := E_1 \parallel x_2 := E_2$, where x_1, x_2 are simply items within the list $x \in \widehat{X}(\ell)$.

Part 1:

Case $C^\ell \triangleq x_1 := E_1 \,[]\, x_2 := E_2$, then:

(Note that x_1, x_2 and x_1', x_2' are members of list x and x' respectively.)

$$
\begin{aligned}
prd_{x,y}(x_1 := E_1[]x_2 := E_2) &\Leftrightarrow \neg[x_1 := E_1[]x_2 := E_2](x', y' \neq x, y) \\
&\Leftrightarrow \neg([x_1 := E_1\ []\ x_2 := E_2](x', y' \neq x, y)) \\
&\Leftrightarrow \neg([x_1 := E_1](x'_1, y' \neq x_1, y) \wedge [x_2 := E_2](x'_2, y' \neq x_2, y)) \\
&\Leftrightarrow \neg((x'_1, y' \neq E_1, y) \wedge (x'_2, y' \neq E_2, y)) \\
&\Leftrightarrow (x'_1, y' = E_1, y) \vee (x'_2, y' = E_2, y) \\
&\Leftrightarrow (y' = y) \ \wedge \ ((x'_1 = E_1) \vee (x'_2 = E_2))
\end{aligned}
$$

Again, we see in this case that for all $y \in X^c$, $\Sigma \equiv^y \Sigma'$ holds.

Part 2: By induction on part 1, part 2 trivially holds.

The foregoing proof of Theorem 3 shows that whenever a variable y is not updated in a substitution S, then y is *independent* of any updated variable x in all stores of S. Thus y must be equivalent in all stores of S, i.e.,

$$\forall x \in \widehat{X}(\ell),\ \forall y \in X^c,\ \forall \sigma_1 \in \Sigma, \sigma_2 \in \Sigma',\ x\widehat{D}(\ell)y \ \Rightarrow\ \sigma_1(y) = \sigma_2(y).$$

Since it is straightforward to extend the proofs given above to other GSL constructs by structural induction, we now look at a related aspect of correctness of our information flow analysis framework, the semantic significance of which is that stores equivalent with respect to the set of variables on which an updated variable(s) depend(s) in a machine are also equivalent with respect to the updated variable(s) on termination of the machine. That is, the after-values of the updated variable(s) are the same in both stores if the sets of variables the updated variable(s) depend on are equivalent in both stores.

This semantic essence is recorded in Theorem 4 below. (Note: we write $\approx^x$ to abbreviate $\approx^{\{x\}}$):

Theorem 4 (Store Independence (GSL))**.** *Suppose* $(\widehat{G}, \widehat{D}) \vDash C^\ell$*, given that* $D_x = \widehat{D}(l)(x)$ *denotes the set of variables upon which* x *depends at* l*, and* $\approx^x$ *denotes equivalence on sets of stores wrt* x*, then, for all* $x \in \widehat{X}(\ell)$*:*

if $(\sigma_1 \approx^{D_x} \sigma_2)$ then $\forall y \in (D_x \cup X^c),\ prd_{x,y}(C^\ell) \ \Rightarrow\ \sigma'_1 \approx^x \sigma'_2$

PROOF:

Given that x ranges over the variables updated in the substitution C^ℓ, i.e., $x \in \widehat{X}(\ell)$; and y ranges over independent variables in C^ℓ, i.e., $y \in X^c$. Note that since $x \in \widehat{X}(\ell)$ and X^c are *mutually exclusive*, clearly $\forall y \in X^c$ if $\Sigma_1 \approx^{X^c} \Sigma_2$ then $\Sigma_1' \approx^x \Sigma_2'$ with respect to X^c. Hence, it suffices to prove Theorem 4 with respect to $y \in D_x$ as shown below. Now, assume $\forall \sigma_1 \in \Sigma, \sigma_2 \in \Sigma,\ x\widehat{D}(\ell)y \ \Rightarrow\ \sigma_1(y) = \sigma_2(y)$. We prove Theorem 4 by cases on the shape of C^ℓ.

Case *skip* : Trivially holds since $D_x \supseteq Id$ - - Table 3.4b

Case $(x := E)^\ell$: Trivial since $D_x \supseteq Id[x \mapsto FV(E)]$ and
$\forall y \in FV(E),\ \Sigma_1 \approx^{FV(E)} \Sigma_2 \ \Rightarrow\ \sigma_1(y) = \sigma_2(y)$ - - Table 3.4b

Case $(\ {}_aC_1^{\ell_1}\ []\ {}_aC_2^{\ell_2})^\ell$: Let $C^\ell \ \equiv\ (\ {}_aC_1^{\ell_1}\ []\ {}_aC_2^{\ell_2})^\ell$
Now, suppose by contradiction that:
$x\widehat{D}(\ell)y \ \Rightarrow\ \sigma_1(y) \neq \sigma_2(y)$ and $prd_{x,y}(C^\ell) \ \not\Rightarrow\ \sigma_1' \approx^x \sigma_2'$.
From Table 3.4b, we recall that our flow analysis of $(\ {}_aC_1^{\ell_1}\ []\ {}_aC_2^{\ell_2})^\ell$ yields $\widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup \widehat{D}(\ell_2)$. For $1 \leq i \leq n$ ranging over labels, we generalise this over a finite (bounded) choice between n substitutions, i.e., $\widehat{D}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{D}(\ell_i)$.
Given that $\sigma_1', \sigma_2' \in \Sigma'$. It follows that for all ${}_aC_i^{\ell_i}$:
$x\widehat{D}(\ell_i)y \ \Rightarrow\ \sigma_1'(y) \neq \sigma_2'(y)$. Contradiction, by Theorem 3.
Also, recall that $prd_{x,y}(C) \Leftrightarrow \neg[C](x, y \neq x', y')$.
Now if $prd_{x,y}(C^\ell) \ \not\Rightarrow\ \sigma_1' \approx^x \sigma_2'$, it follows that:
$prd_{x,y}(C^\ell) \ \Rightarrow\ [C](x, y \neq x', y')$.
This too is a contradiction by the definition of $prd_{x,y}(C^\ell)$ - - Formula 3.2

Thus $\forall y \cdot x\widehat{D}(\ell)y$, if $(\sigma_1 \approx^{D_x} \sigma_2)$ then $prd_{x,y}(C^\ell) \ \Rightarrow\ \sigma_1' \approx^x \sigma_2'$

Case (**IF** b_1 **THEN** $C_1^{l_1}$ **ELSIF** b_2 **THEN** $C_2^{\ell_2}$...**ELSE** $C_n^{\ell_n}$ **END**)$^\ell$:
If $x \notin \widehat{X}(\ell)$, the result follows by assignment freedom (Theorem 3).
If $x \in \widehat{X}(\ell)$: Recall that $\widehat{D}(\ell) \ \supseteq\ (\widehat{X}(\ell) \times (\bigcup_{i=1}^{n-1} FV(b_i)))$. Hence, for all $y \in FV(b_i)$, If $x \in \widehat{X}(\ell)$, then $x\widehat{D}(\ell)y$. That is, the values of y in σ_1 and σ_2 are equal (i times, derivations for $prd_{x,y}(C^\ell)$).

Note: It is straightforward to extend the proofs given here to other GSL constructs by structural induction on the abstract syntax tree.

Our analysis framework also prevents termination-sensitive flows, which are often caused when the termination of a substitution is conditioned on a high security variable such as when a high security variable is used in the precondition of a protected substitution (or in the *while* condition, in implementations). The proof of correctness of this aspect of our analysis, by cases where substitutions executed in some sets of states can be shown to always terminate, is presented in Theorem 5. First, though, we introduce Lemma 3, which we use in the proof of Theorem 5.

Lemma 3 (Termination of Flow Analysis Framework)**.** *For all* C^ℓ *such that* $(\widehat{G},\ \widehat{D})\ \vDash\ C^\ell$*, there is always a least fixed point.*

As noted in Section 3.4, the information flow analysis framework presented in this thesis is parametric on the collection of *finite* variables that may be assigned to (i.e., $\widehat{X}(\ell)$), the collection of *finite* variables that may affect termination of the statement with label ℓ (i.e., $\widehat{G}(\ell)$), and a collection of *finite* dependency pairs in a substitution S (i.e., $\widehat{D}(\ell)$). Consequently, the analysis always has a least fixed point, which implies our analysis terminates always and yields a result, a set of dependency pairs or constraints. We illustrate this lemma with the following example:

(**PRE** $p \neq 0$ **THEN**
  (**IF** $p = q$ **THEN**
    $(x := y + w)^{\ell_3}$
  ELSE
    $(z \Longrightarrow x := 0)^{\ell_2}$
  END)$^{\ell_1}$
END)$^{\ell_0}$

We present in Table 3.6 the *finite* set of constraints (using Table 3.4) with respect to our $(\widehat{X}, \widehat{G}, \widehat{D})$ abstraction of the example.

Notice that no matter how many times we iterate over the example as we collect the constraints, the least fixed point of our information flow analysis

Lab	$\widehat{X}$	$\widehat{G}$	$\widehat{D}$
ℓ_3	$\{x\}$	$\varnothing$	$\{x \mapsto y, x \mapsto w\}$
ℓ_2 ℓ_1	$\{x\}$ $\{x\}$	$\varnothing$ $\widehat{G}(\ell_3) \cup \widehat{G}(\ell_2) = \varnothing$	$Id[x \mapsto \varnothing]$ $\widehat{D}(\ell_3) \cup \widehat{D}(\ell_2) \cup FV(p = q) \cup z$ $= \{x \mapsto y, x \mapsto w, x \mapsto p, x \mapsto q, x \mapsto z\}$
ℓ_0	$\{x\}$	$FV(p \neq 0) = \{p\}$	$\{x \mapsto y, x \mapsto w, x \mapsto p, x \mapsto q, x \mapsto z\}$

Table 3.6: GSL Abstraction of Example Flow Analysis ($\widehat{G}$)

is defined by the *finite* set of constraints on ℓ_0, i.e.,

$$(\widehat{X}, \widehat{G}, \widehat{D}) \triangleq (\{x\},\ \{p\},\ \{x \mapsto y, x \mapsto w, x \mapsto p, x \mapsto q, x \mapsto z\}).$$

Theorem 5 (Termination Independence (GSL))**.** *Given that* $(\widehat{G}, \widehat{D}) \vDash C^\ell$*, and writing* $trm(C^\ell), \Sigma$ *to mean that* C^ℓ *executed in any state* $\sigma \in \Sigma$ *always terminates, i.e., Lemma 3 holds, then:*

(1) If $\bullet \notin \widehat{G}(\ell)$, then $trm(C^\ell), \Sigma$.
(2) If $\Sigma_1 \approx^{\widehat{G}(\ell)} \Sigma_2$ then $trm(C^\ell), \Sigma_1 \Leftrightarrow trm(C^\ell), \Sigma_2$

PROOF:
Note that part 1 indicates that no preconditioned substitution (or iteration in implementation) is encountered in the substitutions of interest. Hence proof of part 1 is trivial and straightforward from the semantics, since for all sets of stores, Σ.

$$(\widehat{G}(\ell) = \varnothing) \Rightarrow (trm(C^\ell), \Sigma)$$

For part 2, we assume $\forall y \in \widehat{G}(\ell) \cdot (\Sigma_1 \approx^y \Sigma_2)$. By symmetry, it suffices to prove that $trm(C^\ell), \Sigma_1 \Rightarrow trm(C^\ell), \Sigma_2$. We provide the proof for preconditioned substitution and "if-elsif" substitutions here.

Assume $trm(C^\ell), \Sigma_1$. Then it is left to show that $trm(C^\ell), \Sigma_2$.

Case (**IF** b_1 **THEN** $C_1^{l_1}$ **ELSIF** b_2 **THEN** $C_2^{\ell_2}$...**ELSE** $C_n^{\ell_n}$ **END**)$^\ell$:
If $\bullet \notin \widehat{G}(\ell)$, the result is immediate by part 1.
If $\bullet \in \widehat{G}(\ell)$, then for $1 \leq i < n$, $\widehat{G}(\ell) \supseteq FV(b_i)$. Hence b_i

evaluates to the same value in both sets of stores, $\Sigma_1,\ \Sigma_2$. That is, given that $\Sigma_1 \approx^{\widehat{G}(\ell)} \Sigma_2$ (from definition), it follows from Theorem 3 that

$$\forall y \in \widehat{G}(\ell) \cdot \forall(\sigma_1 \in \Sigma_1 \wedge \sigma_2 \in \Sigma_2) \cdot \sigma_1(y) = \sigma_2(y).$$

Thus the effect of y on all states in both Σ_1 and Σ_2 is the same. Since y determines termination of C^ℓ, and given that $trm(C^\ell), \Sigma_1$, it follows that $trm(C^\ell), \Sigma_2$ also. Hence $trm(C^\ell), \Sigma_1 \ \Leftrightarrow\ trm(C^\ell), \Sigma_2$.

Case (PRE P THEN ${}_aC^{\ell_1}$ END): Let $C^\ell \equiv$ (PRE P THEN ${}_aC^{\ell_1}$ END)$^\ell$

Since $\widehat{G}(\ell) \supseteq FV(P)$, P evaluates to the same value in both sets of stores. If P evaluates to *false*, then C^ℓ may or may not terminate. By structural induction on the abstract syntax tree, part 2 either holds or fails in both sets of stores. If P evaluates to *true*, then, by structural induction, part 2 holds in both sets of stores. Now suppose by contradiction (in either case) that
$trm(C^\ell), \Sigma_1 \ \Leftrightarrow\ \neg trm(C^\ell), \Sigma_2$.
This will mean that, for some $\sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2,\ \sigma_1' \in \Sigma_1'$, and $\sigma_2' \in \Sigma_2'$, $\exists z \in \widehat{G}(\ell)$ such that $\sigma_1'(z) \neq \sigma_2'(z)$.
(Note: Recall that $\widehat{G}(\ell) \supseteq FV(P)$)
This implies, by Theorem 4, that there must be some y such that:
$x\widehat{D}(\ell)y \wedge \sigma_1(y) \neq \sigma_2(y)$
But since $\widehat{D}(\ell) \supseteq \widehat{X}(\ell) \times FV(P)$, we expect for all $y \in FV(P)$ that:
$x\widehat{D}(\ell)y \wedge \sigma_1(y) = \sigma_2(y)$. (Contradiction indeed)

To prove the last aspect of correctness with respect to our information flow analysis, namely: *every program has an acceptable information flow analysis and that the constraints have solutions*, we first discuss the notion of *Model Intersection Property* (MIP), which we will employ in the proof.

Model Intersection Property (aka. Moore Family)

One of the desirable properties of equivalent sets is the *Model Intersection Property*, which basically states that a new model can be generated as an intersection of two known models. That is, given two models $M_1 \vDash A$ and $M_2 \vDash A$, then there is a model M_3 such that $M_3 \vDash A$. Simply put, the *intersection* of a set of models of a system is itself also a model of the system. In

effect, if P denotes predicate symbols, i range over natural numbers between 1 and a finite number n, and t_i are terms, then, formally, we have:

$$M_3 \vDash P(t_1, \ldots, t_n) \;\Leftrightarrow\; M_1 \vDash P(t_1, \ldots, t_n) \wedge M_2 \vDash P(t_1, \ldots, t_n)$$

Expressed in terms of complete lattices, a subset Y of a complete lattice $\mathcal{L} = (\mathcal{L}, \leq)$ is a Moore family (i.e., satisfies the model intersection property) if and only if

$$\forall Z \subseteq Y \;\cdot\; \bigsqcap Z \in Y$$

This shows that there is a *least* subset of Y that satisfies the properties of complete lattices. This is a useful property that we will use to show that we can always get a *least* analysis that satisfies our information flow analysis framework for GSL.

Given that D_1, D_2, D_3 denote, the finite domains of the models M_1, M_2, M_3, respectively, an important property of the model intersection property is that if $M_3 = M_1 \cap M_2$, then $D_3 = D_1 \times D_2$. This yields the interesting conclusion that models generated using the model intersection property tend to be more efficient due to the fact that they cover a larger domain than the individual component models[2] [64]. Thus, whenever it is that a framework has the model intersection property, then one can be confident that *there, always, is an acceptable model for the framework* [36].

Theorem 6 (Model Intersection Property (GSL))**.** *Given* $(\widehat{G}_1, \widehat{D}_1) \vDash C^\ell$*, and* $(\widehat{G}_2, \widehat{D}_2) \vDash C^\ell$*, we assert that* $(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C^\ell$ *and for all substitution,* S*, the set* $\{(\widehat{G}, \widehat{D}) \mid (\widehat{G}, \widehat{D} \vDash S\}$ *is a Moore family, i.e.:*

$$(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C^\ell \;\Leftrightarrow\; (\widehat{G}_1, \widehat{D}_1) \vDash C^\ell \wedge (\widehat{G}_2, \widehat{D}_2) \vDash C^\ell$$

Notice that Theorem 6 basically states that the greatest lower bound of a set of acceptable models is itself also an acceptable model.

[2] It is a well known fact that models with larger domains give better guidance in theorem proving than models with smaller domains, since the chances of deriving invalidating counterexamples is reduced.

Proof.
Using $\widehat{G}_\top$ and $\widehat{D}_\top$ to denote the top elements for the components of our analysis and employing the approach in [36], we define these components as follows:

$$\widehat{G}_\top = \lambda x \cdot \mathbf{Ide}$$

$$\widehat{D}_\top = \lambda x \cdot \mathbf{Ide} \times \mathbf{Ide}$$

Note: Since Clark et al has given the model intersection property proofs for the commands in their imperative *While* language in [36], we concentrate our proofs here only on those substitutions that do not correspond to any of the ones defined in [36].

By straightforward structural induction on Theorem 6, we can see that $(\widehat{G}_\top, \widehat{D}_\top) \vDash C^\ell$, since for all substitution, C^ℓ, we have $\{(\widehat{G}, \widehat{D}) \mid (\widehat{G}, \widehat{D}) \vDash C^\ell\}$.

Now, it is sufficient to show in all cases, without loss of generality, that

$$(\widehat{G}_1, \widehat{D}_1) \vDash C^\ell \wedge (\widehat{G}_2, \widehat{D}_2) \vDash C^\ell \;\Rightarrow\; (\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C^\ell$$

In a number of the following cases we will be employing the set-theoretic property of distributivity of some symbol, e.g., *union* ($\cup$) over intersection ($\cap$). For clarity, we illustrate in Figure 3.2 the correctness of the axiom (where $\cap$ denotes $\sqcap$):

$$A_1 \cup B_1 \cap A_2 \cup B_2 \supseteq (A_1 \cap A_2) \cup (B_1 \cap B_2),$$

We now begin with the proof case for multiple substitution.

Case $((\mathbf{x_1} \coloneqq \mathbf{E_1})^{\ell_1} \parallel (\mathbf{x_2} \coloneqq \mathbf{E_2})^{\ell_2})^\ell$**:** By induction hypothesis, for $i = 1, 2$

$$(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C_i^{\ell_i}$$

Recall, from Table 3.4b, that $\widehat{D}(\ell) \supseteq Id[x_1 \mapsto FV(E_1)] \cup Id[x_2 \mapsto FV(E_2)]$,

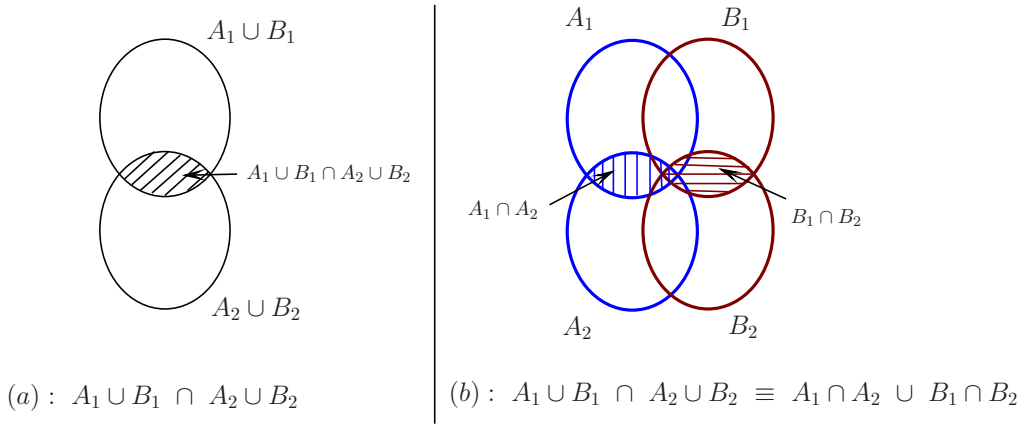


Figure 3.2: Model Intersection Illustration

we have:

$$\begin{aligned}\widehat{D}_1(\ell) \cap \widehat{D}_2(\ell) \supseteq \ & (\widehat{D}_1(\ell_1) \cup \widehat{D}_1(\ell_2)) \ \cap \ (\widehat{D}_2(\ell_1) \cup \widehat{D}_2(\ell_2)) \\ & \text{(by distributivity of union over intersection)} \\ \supseteq \ & (\widehat{D}_1(\ell_1) \cap \widehat{D}_2(\ell_1)) \ \cup \ (\widehat{D}_1(\ell_2) \cap \widehat{D}_2(\ell_2)) \qquad \square\end{aligned}$$

Next, we give the proof case for guarded substitution.

Case $(\mathbf{Q} \Longrightarrow \mathbf{C_1^{\ell_1}})^{\ell}$**:** By the induction hypothesis, for $i = 1, \ 2$

$$(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C_1^{\ell_1}$$

Recall that from Table 3.4b, for multiple substitution, $\widehat{G}(\ell) \supseteq \widehat{G}(\ell_1)$. Hence, we have:

$$\widehat{G}_1(\ell) \cap \widehat{G}_2(\ell) \supseteq \ (\widehat{G}_1(\ell_1) \cap \widehat{G}_2(\ell_1)) \qquad \text{- trivially holds}$$

Similarly, since from Table 3.4b, $\widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(Q))$, we have:

$$\begin{aligned}\widehat{D}_1(\ell) \cap \widehat{D}_2(\ell) \supseteq \ & (\widehat{D}_1(\ell_1) \cup (\widehat{X}(\ell) \times FV(Q))) \ \cap \\ & (\widehat{D}_2(\ell_1) \cup (\widehat{X}(\ell) \times FV(Q))) \\ & \text{(by distributivity of union over intersection)} \\ \supseteq \ & (\widehat{D}_1(\ell_1) \cap \widehat{D}_2(\ell_1)) \ \cup \ (\widehat{X}(\ell) \times FV(Q)) \qquad \square\end{aligned}$$

We now give the proof case for protected substitution, which has the potential to abort.

Case $(\mathbf{P} \mid \mathbf{C_1^{\ell_1}})^{\ell}$**:** By the induction hypothesis, for $i = 1,\ 2$

$$(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C_1^{\ell_1}$$

In the case when $\bullet \notin \widehat{G}(\ell)$, then from Table 3.4b, $\widehat{G}(\ell) \supseteq \widehat{G}(\ell_1)$. Hence, we have:

$$\widehat{G}_1(\ell) \cap \widehat{G}_2(\ell) \supseteq\ (\widehat{G}_1(\ell_1) \cap \widehat{G}_2(\ell_1)) \qquad \text{- trivially holds}$$

Also, from Table 3.4b, $\widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(P))$, so by induction from case $Q \Longrightarrow C^{\ell_1}$, we have:

$$\widehat{D}_1(\ell) \cap \widehat{D}_2(\ell)\ \supseteq\ (\widehat{D}_1(\ell_1) \cap \widehat{D}_2(\ell_1))\ \cup\ (\widehat{X}(\ell) \times FV(P))$$

However, in the case when $\bullet \in \widehat{G}(\ell)$, then, by Table 3.4b,

$$(\widehat{G}_1 \sqcap \widehat{G}_2) \supseteq FV(P)$$

Thus:

$$(\widehat{G}_1 \sqcap \widehat{G}_2)(\ell_1) \supseteq \{\bullet\}$$

$$\widehat{G}_1(\ell_1) \supseteq \{\bullet\} \text{ or } \widehat{G}_2(\ell_1) \supseteq \{\bullet\}, \text{ or both.}$$

Now, if

$$Y = \{(\widehat{G}, \widehat{D}) \mid (\widehat{G}, \widehat{D})\ \vDash\ C^{\ell}\},$$

Then, for all $i = 1,\ 2$:

$$Y = \{(\widehat{G}_i, \widehat{D}_i) \mid (\widehat{G}_i, \widehat{D}_i)\ \vDash\ C^{\ell}\} \qquad \square$$

We follow with the proof case for bounded choice.

Case $(\mathbf{C_1^{\ell_1}[]C_2^{\ell_2}})^{\ell}$**:** By the induction hypothesis, for $i = 1, 2$
$(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C_i^{\ell_i}$

Since from Table 3.4b, for the choice operation, $\widehat{G}(\ell) \supseteq \widehat{G}(\ell_1) \cup \widehat{G}(\ell_2)$, we have:

$$\begin{aligned}\widehat{G}_1(\ell) \cap \widehat{G}_2(\ell) &\supseteq\ (\widehat{G}_1(\ell_1) \cup \widehat{G}_1(\ell_2))\ \cap\ (\widehat{G}_2(\ell_1) \cup \widehat{G}_2(\ell_2)) \\ &\qquad \text{(by distributivity of union over intersection)} \\ &\supseteq\ (\widehat{G}_1(\ell_1) \cap \widehat{G}_2(\ell_1))\ \cup\ (\widehat{G}_1(\ell_2) \cap \widehat{G}_2(\ell_2))\end{aligned}$$

Similarly, since from Table 3.4b, $\widehat{D}(\ell) \supseteq \widehat{D}(\ell_1) \cup \widehat{D}(\ell_2)$, we have:

$$\begin{aligned}\widehat{D}_1(\ell) \cap \widehat{D}_2(\ell) &\supseteq\ (\widehat{D}_1(\ell_1) \cup \widehat{D}_1(\ell_2))\ \cap\ (\widehat{D}_2(\ell_1) \cup \widehat{D}_2(\ell_2)) \\ &\qquad \text{(by distributivity of union over intersection)} \\ &\supseteq\ (\widehat{D}_1(\ell_1) \cap \widehat{D}_2(\ell_1))\ \cup\ (\widehat{D}_1(\ell_2) \cap \widehat{D}_2(\ell_2))\end{aligned}$$ □

Case $(\mathbf{IF}\ b_1\ \mathbf{THEN}\ C_1^{\ell_1}\ \mathbf{ELSIF}\ b_2\ \mathbf{THEN}\ C_2^{\ell_2}\ \ldots\ \mathbf{ELSE}\ C_n^{\ell_n}\ \mathbf{END})^{\ell}$**:** Since our $\widehat{G}$ and $\widehat{D}$ components operate on GSL, and hence $\widehat{G}$ and $\widehat{D}$ are finite, it is safe to restrict **Ide** and **Lab** to the identifiers and labels appearing in the program. Hence, we assume some *finite* natural number n where i range over $\{1..n\}$. By the induction hypothesis, for i such that $0 < i \leq n$,
$(\widehat{G}_1 \sqcap \widehat{G}_2, \widehat{D}_1 \sqcap \widehat{D}_2) \vDash C_i^{\ell_i}$

In the case when $\bullet \notin \widehat{G}(\ell)$, then from Table 3.4b, $\widehat{G}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{G}(\ell_i)$
For all i such that $1 < i \leq n$,

$$\begin{aligned}\widehat{G}_{i-1}(\ell) \cap \widehat{G}_i(\ell) &\supseteq\ (\widehat{G}_{i-1}(\ell_{i-1}) \cup \widehat{G}_{i-1}(\ell_i))\ \cap\ (\widehat{G}_i(\ell_{i-1}) \cup \widehat{G}_i(\ell_i)) \\ &\qquad \text{(by distributivity of union over intersection)} \\ &\supseteq\ (\widehat{G}_{i-1}(\ell_{i-1}) \cap \widehat{G}_i(\ell_{i-1}))\ \cup\ (\widehat{G}_{i-1}(\ell_i) \cap \widehat{G}_i(\ell_i))\end{aligned}$$

Also, from Table 3.4b, $\widehat{D}(\ell) \supseteq \bigcup_{i=1}^{n} \widehat{D}(\ell_i) \cup (\widehat{X}(\ell) \times \bigcup_{i=1}^{n-1} FV(b_i))$

By induction on the case for $\widehat{G}_{i-1}(\ell) \cap \widehat{G}_i(\ell)$, we can safely conclude that

$\widehat{D}_{i-1}(\ell) \sqcap \widehat{D}_i(\ell)$ also holds for all i such that $1 < i < 2n$ - (since total $\widehat{D}$ pairs $= 2n - 1$).

However, in the case when $\bullet \in \widehat{G}(\ell)$, then, by Table 3.4b,

$$(\widehat{G}_1 \sqcap \widehat{G}_2) \supseteq \bigcup_{i=1}^{n-1} FV(b_i), \text{ and for all } i \text{ such that } 1 < i \leq n\text{:}$$

$$(\widehat{G}_{i-1} \sqcap \widehat{G}_i)(\ell_{i-1}) \supseteq \{\bullet\} \ \vee \ (\widehat{G}_{i-1} \sqcap \widehat{G}_i)(\ell_i) \supseteq \{\bullet\}$$

Thus, either:

$$\widehat{G}_{i-1}(\ell_{i-1}) \supseteq \{\bullet\} \ \vee \ \widehat{G}_i(\ell_{i-1}) \supseteq \{\bullet\}$$

or

$$\widehat{G}_{i-1}(\ell_i) \supseteq \{\bullet\} \ \vee \ \widehat{G}_i(\ell_i) \supseteq \{\bullet\}$$

or both. Now suppose

$$Y = \{(\widehat{G}, \widehat{D}) \mid (\widehat{G}, \widehat{D}) \ \vDash \ C^\ell\},$$

Then, for all i such that $1 < i \leq n$:

$$Y = \{(\widehat{G}_i, \widehat{D}_i) \mid (\widehat{G}_i, \widehat{D}_i) \ \vDash \ C^\ell\},$$

i.e.,

$$\bigsqcap_{i=1}^{n} Y = \{(\widehat{G}_i, \widehat{D}_i) \mid (\widehat{G}_i, \widehat{D}_i) \ \vDash \ C^\ell\}$$

And this we can rewrite as:

$$\bigsqcap_{i=1}^{n} Y = (\widehat{G}_\top, \widehat{D}_\top) \ \sqcap \ (\widehat{G}_1, \widehat{D}_1) \ \sqcap \cdots \sqcap \ (\widehat{G}_n, \widehat{D}_n) \qquad \square$$

Note: The **SELECT**...**WHEN** substitution can be translated into an **IF**...**ELSIF** substitution, hence there is no need to prove the model intersection property with respect to the **SELECT**...**WHEN** substitution. Also, the proof case for other substitutions defined in our GSL semantics, such as unbounded choice (**ANY**) substitutions, can be derived as a combination of some of the proofs already given here.

With the foregoing, we have proved, using the model intersection property, that a *least* analysis exists for our information flow analysis framework for GSL, hence, it is always possible to derive acceptable information flow analysis solutions to any well-formed GSL[3]. The fact that a least analysis exists for our analysis framework also proves that the analysis *terminates*. Clark et al [36] suggested for further investigation the matter of whether more efficient solutions could be found for their information flow analysis framework. We suggest one such efficient solution in Section 3.5 where we enhance our information flow analysis framework for GSL with a variant of *Reaching Definitions Analysis* that we term *Reaching Dependencies Analysis*. In the following subsection, though, we extend the flow analysis given for the *while* command in [36] to the *substitution* in B implementation.

3.4.2 Flow Analysis of Implementation Substitutions

We adopt the information flow analysis of sequential commands given by Clark et al in [36], since it is analogous to the flow analysis of sequential substitutions in B implementations. The same is true for **IF... ELSE... END** substitutions. However, since the definition of a *while* substitution in B implementation is slightly different from the *while* command defined in [36], we present below a B *while* substitution and its flow analysis.

The definition of B *while* substitution differs from the one given by [36] only in that two additional keywords: **INVARIANT** and **VARIANT** are added to the syntax. The only extra work required to analyse B *while* substitutions is the addition of flows from the free variables in the **INVARIANT**, J, and the **VARIANT**, V, to our flow constraints, as shown below. Hence, using the notation given in Table 3.1, we simply present without proof the flow analysis for B *while* substitutions, since the proof given by Clark et al in [36] is sufficient.

$$(\widehat{G}, \widehat{D}) \vDash (\textbf{WHILE}\ b\ \textbf{DO}\ C_1^{\ell_1}\ INVARIANT\ J\ VARIANT\ V)^{\ell}$$

[3]Notice in the given derivations that the conclusion in each case of the GSL constructs shown above, the resulting constraints definition is an acceptable model. (Compare derivation results with corresponding entries in Table 3.4b for each construct.)

$$
\begin{array}{l}
\Leftrightarrow (\widehat{G}, \widehat{D}) \ \vDash \ C_1^{\ell_1} \ \wedge \\
\quad \widehat{G}(\ell) \supseteq \{\bullet\} \ \cup \ FV(b) \ \cup \ \widehat{G}(\ell_1) \ \cup \ \widehat{G}(\ell); \widehat{D}(\ell_1) \ \cup \\
\qquad FV(J) \ \cup \ FV(V) \ \wedge \\
\quad \widehat{D}(\ell) \supseteq Id \cup (\widehat{D}(\ell); \widehat{D}(\ell_1)) \ \cup \ (\widehat{X}(\ell) \times FV(b)) \\
\qquad (\widehat{X}(\ell) \times FV(J)) \ \cup \ (\widehat{X}(\ell) \times FV(V))
\end{array}
$$

3.5 Optimisation of GSL Flow Analysis with Reaching Dependencies Analysis

There are often cases where a variable $x \in \widehat{X}$ is redefined within a program. For example, in the program: $x := 7; y := 1; \mathit{while}(x > 1)\ \mathit{do}\ y := x * y\ ;\ x := x - 1\ \mathit{end}$, which calculates the factorial of x, on first entry into the loop, y takes the value 1, whereas on subsequent iterations, the value of y carried over from the assignment $y := 1$ is redefined as $x * y$. In such cases, whenever we determine during the *parsing* of the substitution's abstract syntax tree that a variable $x \in \widehat{X}$ is redefined (or updated) within a sequence of substitutions, we can optimise our information flow analysis for GSL by only collecting the last of such redefined pairs of dependencies in the $\widehat{D}$ component of our framework. This will enable us to still correctly analyse secure programs, but without collecting *redundant* constraints in $\widehat{D}$. Notice that this approach, rather than being an alternative, is an enhancement of the flow analysis framework introduced in this thesis.

We pointed out in Section 2.2.2.10 that the flow analysis approach introduced by Clark et al [36], on which we base our framework in this thesis, is flow sensitive, i.e., programs that could otherwise be judged insecure by many security type systems are correctly judged secure. For example, given a lattice ordering where $y \not\sqsubseteq x$, the substitution $x := y + 1; x := 0$ will be correctly analysed as secure, since the secret flow from y into x has been overwritten by a subsequent substitution to x, which is secure. Now, rather than collect the $\widehat{D}$ constraint for $x := y+1$, only to overwrite it later with the $\widehat{D}$ constraint for $x := 0$, our proposed optimisation approach will make the analysis *leaner*, by ignoring the dependencies of overwritten variables and only collecting the $\widehat{D}$ for the last instance where the variable is redefined. Hence, in this example, without loss of the flow-sensitivity property of our

flow analysis framework, we will simply collect only the $\widehat{D}$ constraint for $x := 0$. For another example, suppose we have a two-level security lattice, $\mathcal{L} = \langle \mathcal{L}, \sqsubseteq \rangle$, where $H, L \in \mathcal{L} \cdot H \not\sqsubseteq L$, and $x, y, z \in \mathbf{Ide} \cdot \{x \mapsto L, y \mapsto L, z \mapsto H\}$. Again, notice that, based on this security classification, the following program is secure although the subprogram $y := z + 1$ is insecure.

$x := 7;$
$y := z + 1;$
while$(x > 1)$ **do**
    $y := y * x;$
    $x := x - 1$
end;
$y := 0$

(Note: A high security variable, z, flows into a low security variable, y, in the second assignment, and this value can be deduced by backtracking the final value of y on termination of the loop, if there is no further assignment to y. However, the value of y is later overwritten by the assignment $y := 0$, hence the output of y in all runs of the program, in an *imperative, noninteractive uniprocessing setting*, is always 0. Thus, the overall program is secure.)

Notice, though, that the variable y is updated three times in the program: once before the *while* loop, once within the *while* loop, and once after. Clearly, whatever *insecure* information might have *interfered* with y in the first two redefinitions of y will have been overwritten by the last substitution to y before the program terminates. So, in our setting where parallel processing of subprograms is not allowed, we can save some work by only collecting the $\widehat{D}$ constraint for $y := 0$ in this example, which is $\varnothing$ (provided that y has not been saved internally or copied to another variable). Notice here, too, that the removal of redefined $\widehat{X}$ members from our $\widehat{D}$ constraints does not limit the flow-sensitivity functionality of our analysis framework.

This approach of removing $\widehat{D}$ pairs for overwritten variables, termed *Reaching Dependencies Analysis*, relates closely to the notion of *Reaching Defini-*

tions Analysis[4] as discussed by Hankin and the Nielsons in [64]. *Reaching Definitions Analysis* is built on the notions of *elementary program blocks* and *program points*. The term 'elementary blocks' refer to assignments, tests, and skip statements [64], which constitute the most elementary building blocks of programs. These elementary program blocks are individually labeled, and written within square brackets. Hence, Listing 3.8 below shows a program broken into its elementary program blocks:

$$[x := 7]^{\ell_1}; [y := 1]^{\ell_2} \ ; \ while [x > 1]^{\ell_3} \ do \ [y := x * y]^{\ell_4} \ ; \ [x := x - 1]^{\ell_5} \ end \quad (3.8)$$

In our formalism of *Reaching Dependencies Analysis*, however, we use the term *semantic program blocks* to refer to the GSL semantics defined in Table 3.2 earlier. The motivation for this is because the GSL semantic blocks in Table 3.2 constitute the smallest blocks for analysing possibilistic information flow in the GSL setting. Further, we aim to develop an optimisation of our information flow analysis framework, not an alternative framework, hence it is necessary to use the same semantic blocks used in developing our information flow constraints in Table 3.4b. Thus, for example, Listing 3.8, above could be re-written in terms of its semantic program blocks as shown in Listing 3.9 below:

$$[x := 7]^{\ell_1}; [y := 1]^{\ell_2} \ ; \ [while (x > 1) \ do \ [y := x * y]^{\ell_4} \ ; \ [x := x - 1]^{\ell_5} \ end]^{\ell_3} \quad (3.9)$$

The authors [64] used the notion of program point to refer to the *entry* or *exit* to/from an elementary program block since 'Definitions' means 'assignments' (or in GSL terminology: Simple substitutions). Correspondingly, in our framework, a program point refers to the *entry* or *exit* to/from a semantic program block. The intuition here is to, without loss of generality, be able to collect flow constraints in terms of the semantic program blocks of the GSL. Thus, whereas reaching definitions analysis collects, for each program point, assignments (or simple substitutions) that may have been made and not overwritten when program execution reaches that point, our extension of this concept to the notion of *reaching dependencies analysis* collects, for each program point, the sets of pairs of variable dependencies, $\widehat{D}$, where the assigned variable(s) have not been overwritten, thereby cap-

[4] An information flow analysis which statically determines which definitions (or *assignment*) may reach (i.e., not overwritten up to) a given point in a program.

turing only *live* dependencies up to that point. Hence we build the reaching dependencies analysis enhancement of our original framework on the core notion that:

> A semantic program block of the form $[P|@x' \cdot (Q \Longrightarrow x := x')]^{\ell}$ *may reach* a certain program point if there is an execution of the program where x was last assigned a value at ℓ when the program point is reached [64].

This can be semi-formally expressed in terms of two functions, namely:

- $overwrite(x)$, which collects all variables $x \in \widehat{X}$ that are subsequently overwritten within the program, and
- $killRedef(x)$, which removes all redefined variables, leaving only those variables $x \in \widehat{X}$ not redefined in the program for analysis with respect to $\widehat{D}$, ***provided*** *the redefined variable(s) have not been stored first, e.g., as in the swap/overwrite sequence* $x := a + 1; y := x; x := b - 1$, *in which case the variable* a *indirectly flows into* y *before* x *is overwritten, hence we need to collect* $\widehat{D}$ *with respect to the first update of variable* x *also.*

Given that $Blocks$ is a set of all semantic program blocks in the system, $spb_1, spb_2 \in Blocks$ are individual semantic program blocks where spb_1 appears before spb_2 (denoted $spb_1 \supset spb_2$) in the execution sequence, and x', x'' are *fresh* variables, we define $overwrite(x)$ in normal form as follows:

> Given that $\supset^*$ denotes 1 or more sequence(s) of semantic blocks over which i ranges:
>
> $spb_1 = P_1|@x' \cdot (Q_1 \Longrightarrow x = x')$, and
> $spb_2 = P_2|@x'' \cdot (Q_2 \Longrightarrow x = x'')$,
>
> *if*
>
> $spb_1 \supset^* spb_2 \wedge (\forall spb_i \in Blocks \cdot spb_1 \supset spb_i \wedge \nexists y \in \widehat{X} \cdot (y, x) \in \widehat{D})$
>
> *then*
>
> $overwrite(x) \triangleq \{x \mid spb_1 \supset spb_2\}$

To define $killRedef(x)$, for all $x \in \widehat{X}$, we suppose there is a function $last(x)$ that records the last occasion the variable x is rewritten in the sequence

of blocks $spb_1 \supset^* spb_2$. Although we do not provide a formal definition for $last(x)$ here, in practice, while parsing the abstract syntax tree of the program using C++ for example, it is a simple thing to reference the last element in a vector used to incrementally record redefined variables in $\widehat{X}$. Thus we simply define $killRedef(x)$ here as yielding one of the two cases below:

$$\forall x \in \widehat{X},\ killRedef(x) = \begin{cases} \varnothing, & if\ overwrite(x) = \varnothing \\ last(x), & if\ overwrite(x) \neq \varnothing \end{cases}$$

The second case basically records only the dependency pairs $\widehat{D}$ in the last semantic block wherein a variable is overwritten for the last time in a sequence of blocks.

Consequently, writing $RD(\ell)$ to denote the reaching dependencies of the program labeled ℓ, we semi-formally express the notion of reaching dependencies analysis in terms of the functions $overwrite(x)$ and $killRedef(x)$ as follows:

$$RD(\ell) = \widehat{D}(\ell),\ \text{where}\ P|@x' \cdot ((Q \Longrightarrow x := x'') \wedge killRedef(x))$$

For example, in our running example, we say that $[x := 7]^{\ell_1}$ reaches the entry to $[y := 1]^{\ell_2}$. This notion is formally captured as an abstraction, mapping labels to *dependencies*, i.e., for all $\ell \in \mathbf{Lab}$, $RD \in \mathbf{Lab} \to \widehat{D}$. Thus, we more formally write '$(\ell_1, \widehat{D}(\ell_1))$ reaches the entry to ℓ_2'.

Let **Blocks** denote the set of semantic program blocks in a sequence of substitutions S; and let "?" denote uninitialised variables. We introduce some constructor functions for use in the reaching dependencies analysis, but first, we define **Blocks** in terms of S as follows:

$$\mathbf{Blocks} \subseteq \mathcal{P}(S)$$

The functions we introduce here include:

- An initialisation function, $init_{RD}$, which is used to record the set of dependencies at the beginning of a semantic program block. In the case where there are no preceding blocks (or where $x \in \widehat{X}$ is assigned a *constant*), the $init_{RD}$ of the current block defaults to $\varnothing$, otherwise,

if unknown or uninitialised, (e.g. where S is a sub-program of a larger system whose $\widehat{D}$ is unknown to S), it defaults to the special character "?". Formally, we define $init_{RD}$ as follows:

$\forall B^{\ell} \in \mathbf{Blocks}$, $init_{RD} \in \mathbf{Lab} \rightarrow \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$
(since $\widehat{D} \in \mathbf{Ide} \times \mathbf{Ide}$)
i.e., $init_{RD}(\ell) = \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$

➠ The next function, $Kill_{RD}$, records overwritten (or destroyed) substitutions as a powerset of variable dependency pairs for every $x \in \widehat{X}$ overwritten in the labeled program block of interest, thus,

$\forall B^{\ell} \in \mathbf{Blocks}$, $kill_{RD} \in \mathbf{Lab} \rightarrow \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$
i.e., $kill_{RD}(\ell) = \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$

Notice that in our context, the subscript $_{RD}$ refers to *reaching dependencies*, and not reaching definitions as in [64]. The intuition behind our definition is that whenever a new substitution is made to a variable in a block, then we do not need to consider earlier flows to that variable within the program in our analysis, since these dependencies do not propagate to the output of the program, provided the earlier flows to the said variable(s) have not been stored elsewhere. The removal of such overwritten variables and their dependencies, $\widehat{D}$, streamlines our analysis. Formally, given that $B^{\ell_1}, B^{\ell_2} \in \mathbf{Blocks}$ denote semantic program blocks whereby $\ell_1, \ell_2 \in \mathbf{Lab}$; B^{ℓ_1} initially assigns a value to x, and this value was later overwritten within B^{ℓ_2}, we express $kill_{RD}$ as follows:

Given that $S = \ldots B^{\ell_1} \cup B^{\ell_2}$:

$kill_{RD}(\ell_2) = \{(x, ?)\} \cup \widehat{D}(\ell_1)$

➠ We also introduce a function, gen_{RD}, which collects all dependency pairs that are generated by the block under consideration. This function has the same signature as $kill_{RD}$ defined earlier, hence, formally, gen_{RD} is defined as follows:

$\forall B^{\ell} \in \mathbf{Blocks}$, $gen_{RD} \in \mathbf{Lab} \rightarrow \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$
i.e., $gen_{RD}(\ell) = \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$

Given, for example, that $S = [x := y + z]^{\ell_1}; [w := v - 1]^{\ell_2}$, then
$gen_{RD}(\ell_1) = \{(x, y), (x, z)\}$, and $gen_{RD}(\ell_2) = \{(w, v)\}$.

Table 3.7 summarise $kill_{RD}$ and gen_{RD} in terms of GSL semantic program blocks.

➠ The next function on labels that we introduce here, $final_{RD}$, records the set of all possible dependencies after execution of a semantic program block, and this is formally defined, in similar fashion to $init_{RD}$, as follows:

$$\forall B^{\ell} \in \mathbf{Blocks},\ final_{RD} \in \mathbf{Lab} \rightarrow \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$$
$$\text{i.e., } final_{RD}(\ell) = \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$$

The function $final_{RD}$ is computed from the preceding functions as shown below

$$final_{RD} = (init_{RD} - kill_{RD}) \cup gen_{RD}$$

We present a summary of the application of both $init_{RD}()$ and $final_{RD}()$ to GSL semantic blocks in Table 3.8

To complete our information flow analysis we require a *transfer function*, which will need to operate on flows between GSL statements (or, more specifically, semantic program blocks). Given that **Stmt** denotes a set of GSL statements, such that for all GSL substitutions, S, $S \in \mathbf{Stmt}$; and writing $flow(S)$ to denote the transfer function between sub-statements of the GSL S, we model the function *flow* as:

$$\forall S \in \mathbf{Stmt},\ flow \in \mathbf{Stmt} \rightarrow \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$$
$$\text{i.e., } flow(S) = \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$$

It must be acknowledged that further work is required to hone our proposed reaching dependencies analysis optimisation of the information flow framework discussed in this thesis. However, the intuition behind the approach and the initial work reported in this thesis is promising. In the meantime, we focus in the next section on the development of security conditions for GSL specifications and refinements based on our core information flow analysis framework.

Killed/Generated Dependencies		
Semantic Block	$\mathbf{kill_{RD}}()$	$\mathbf{gen_{RD}}()$
$[skip]^{\ell}$	$\varnothing$	$\varnothing$
$[x := E]^{\ell_2}$	$\{(x,?)\} \cup \widehat{D}(\ell_1)$	$\{(x,v) \mid v \in \mathbf{Ide} \wedge v \in FV(E)\}$
$[Q \Longrightarrow C]^{\ell}$	$kill_{RD}(\ell)$	$gen_{RD}(\ell)$
$[P \mid C]^{\ell}$	$kill_{RD}(\ell)$	$gen_{RD}(\ell)$
$[@z \cdot (z \in U) \Longrightarrow C]^{\ell}$	$kill_{RD}(\ell)$	$gen_{RD}(\ell)$
$[@z \cdot Q \Longrightarrow C]^{\ell}$	$kill_{RD}(\ell)$	$gen_{RD}(\ell)$
$[\mathbf{IF}\ b\ \mathbf{THEN}\ C_1^{\ell_1}\ \mathbf{ELSE}\ C_2^{\ell_2}]^{\ell}$	$kill_{RD}(\ell_1) \cup kill_{RD}(\ell_2)$	$gen_{RD}(\ell_1) \cup gen_{RD}(\ell_2)$
$[C_1^{\ell_1}[]C_2^{\ell_2}]^{\ell}$	$kill_{RD}(\ell_1) \cup kill_{RD}(\ell_2)$	$gen_{RD}(\ell_1) \cup gen_{RD}(\ell_2)$

Table 3.7: Killed and Generated Dependencies

3.6 Security Conditions for GSL Specifications and Refinements

We discussed noninterference and its extension to generalised noninterference (to deal with secure information flow in nondeterministic and/or nonterminating systems) in Section 2.2.2. In this section, we introduce security conditions that guarantee generalised noninterference in the presence of *underspecification*, and correspondingly, noninterference in deterministic systems. We present here our derived semantic and syntactic security conditions and show how each relates to the other.

We introduce a security lattice, $\mathcal{L} = \langle \mathcal{L}, \leq \rangle$, which we view as having a *weak ordering* attribute on non-empty sets of security classes, i.e., an ordering that is transitive, reflexive and antisymmetric, while also containing both a minimal[5] and a maximal[6] element. For example, writing $\{\bot\}$ to denote the minimal element and $\{\top\}$ the maximal element, with intermediate classes

[5]Minimal element, $\bot \in \mathcal{L}$: $\forall l \in \mathcal{L}$, $(l \leq \bot) \Rightarrow (l = \bot)$.
[6]Maximal element, $\top \in \mathcal{L}$: $\forall l \in \mathcal{L}$, $(l \geq \top) \Rightarrow (l = \top)$.

Initial/Final Dependencies		
Semantic Block	$\mathbf{init_{RD}}()$	$\mathbf{final_{RD}}()$
$[skip]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell)$
$[x := E]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell) \cup \widehat{D}(\ell)$
$[Q \Longrightarrow C^{\ell_1}]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell_1) \cup \widehat{D}(\ell)$
$[P \mid C^{\ell_1}]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell_1) \cup \widehat{D}(\ell)$
$[@z \cdot (z \in U) \Longrightarrow C^{\ell_1}]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell_1) \cup \widehat{D}(\ell)$
$[@z \cdot Q \Longrightarrow C^{\ell_1}]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell_1) \cup \widehat{D}(\ell)$
$[\mathbf{IF}\ b\ \mathbf{THEN}\ C_1^{\ell_1}\ \mathbf{ELSE}\ C_2^{\ell_2}]^{\ell}$	$\varnothing$ or ?	$init_{RD}(\ell) \cup final_{RD}(\ell_1) \cup final_{RD}(\ell_2)$
$[C_1^{\ell_1}[]C_2^{\ell_2}]^{\ell}$	$init_{RD}(\ell_1) \cup init_{RD}(\ell_2)$	$init_{RD}(\ell) \cup final_{RD}(\ell_1) \cup final_{RD}(\ell_2)$

Table 3.8: Initial and Final Dependencies

denoted $\{a\}$ and $\{b\}$, a security lattice on these elements, as illustrated in Figure 3.3, can be written as $\langle\{\ \{\bot\}, \{a\}, \{b\}, \{\top\}\ \},\ \leq\rangle$. Basically, $\mathcal{L}$ is a *powerset* lattice. (Note: we write an *ordered pair* between angled brackets.)

Thus, formally:

> Assuming existence of a minimal and a maximal element in a collection C of sets of security classes, we define the security lattice $\mathcal{L}$ and the (binary) ordering relation, $\leq$, on C as:
>
> $$\mathcal{L} \ = \ \mathcal{P}(C)$$
>
> $$\leq \ = \{\langle X, Y\rangle \mid (X \in C \wedge Y \in C)\ X \subseteq Y\}$$

We then define a multilevel secure system by adding *security tags* to program variables in the manner introduced by Denning and Denning [50], [51]. This we do by defining *function mappings* from **Ide** to $\mathcal{L}$ such that variables with the same security classification map to the same security level in $\mathcal{L}$. We

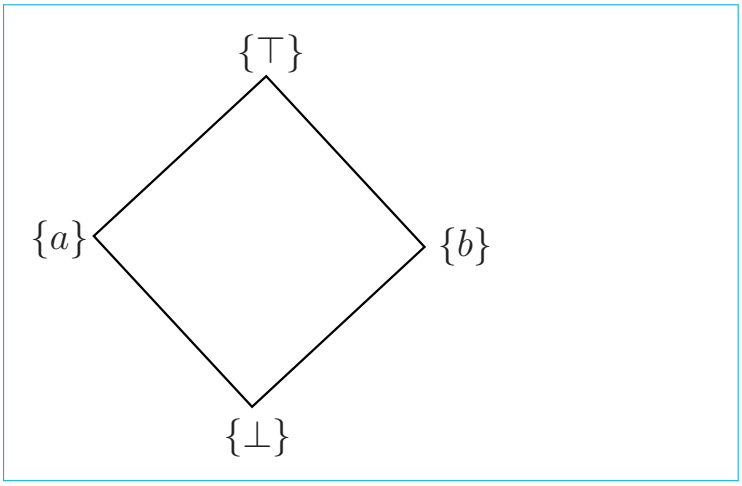


Figure 3.3: Lattice $\langle \{\ \{\bot\}, \{a\}, \{b\}, \{\top\}\ \},\ \leq \rangle$ Illustration

allow for the possibility of some variables not being mapped to any security level, hence, this security mapping function is a *partial function*. Suppose $classify$ denotes such a function and ? denotes *unclassified*, we formally define $classify \in \mathbf{Ide} \rightarrow \mathcal{L}$, as:

$$classify \ = \ \begin{cases} \{(x, l) \mid (x \in \mathbf{Ide} \ \wedge \ l \in \mathcal{L})\ x \mapsto l\}, & \textit{if } x \text{ is classified} \\ \{(x, ?) \mid (x \in \mathbf{Ide} \wedge ? \notin \mathcal{L})\ x \mapsto ?\}, & \textit{if } x \text{ is unclassified} \end{cases}$$

It is easy to see that $classify$ induces an ordering on **Ide**, hence the multilevel secure system causes a *partitioning* of **Ide**. This point is illustrated, for example, in Figure 3.4, where we assume the variables v, w, x, y, z are mapped to the security classes illustrated earlier in Figure 3.3 as shown in Figure 3.4. Formally, suppose the set of all mappings from **Ide** to $\mathcal{L}$ is denoted **SC**, i.e., $\mathbf{SC} = \mathbf{Ide} \times \mathcal{L}$. Given that $x_1 \in \mathbf{Ide}$, $x_2 \in \mathbf{Ide}$, $l_1 \in \mathcal{L}$, and $l_2 \in \mathcal{L}$, we write $flowOrder$ to denote the induced ordering on the domain of **SC**, i.e., **Ide**. First, we define a binary relation, "*securely flows into*", on the domain of **SC**, denoted $\leq_{\mathcal{L}}$, as follows:

$$\leq_{\mathcal{L}} \ = \ \{(x_1, x_2) \mid ((x_1, l_1) \in \mathbf{SC} \ \wedge \ (x_2, l_2) \in \mathbf{SC})\ l_1 \leq l_2\} \qquad (3.10)$$

Thus, $\quad flowOrder = \langle \mathbf{Ide}, \leq_{\mathcal{L}} \rangle$

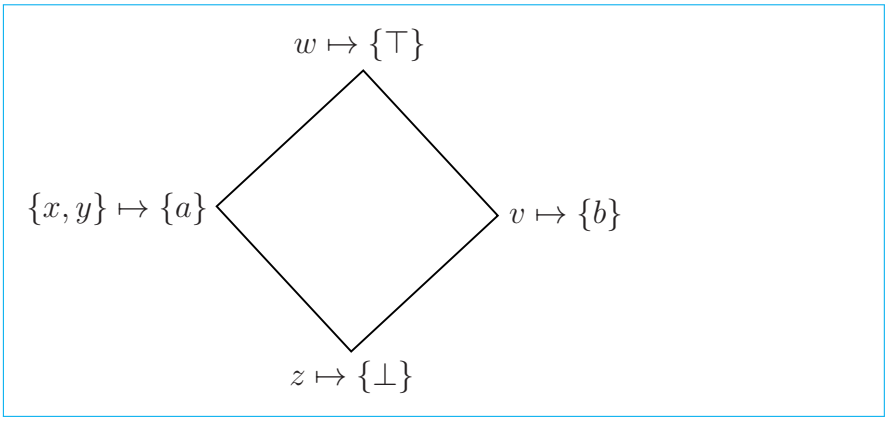


Figure 3.4: (Induced) Partitioning of **Ide** by $\langle\{\ \{\bot\},\{a\},\{b\},\{\top\}\ \},\ \leq\rangle$

Intuitively, $flowOrder$ describes the notion that information may flow from x_1 to x_2, since x_1 is adjudged *lower than or equal to* x_2 by the $\leq_{\mathcal{L}}$ ordering. That is, information in x_1 "*securely flows into*" x_2 if and only if the security class of x_1 is *lower than or equal to* the security class of x_2 (i.e., $l_1 \leq l_2$). Applying this to the example illustrated in Figure 3.4, we have:

$$\textbf{Ide} = \{v, w, x, y, z\}, \text{ and } \mathcal{L} \ = \ \langle\{\ \{\bot\},\{a\},\{b\},\{\top\}\ \},\ \leq\rangle.$$

Thus $\quad flowOrder = \{(z, v), (z, x), (z, y), (z, w), (v, w), (x, w), (y, w)\}$

(**Note:** We use $\not\leq_{\mathcal{L}}$ to denote the contrapositive ordering relation (i.e., dual) of $\leq_{\mathcal{L}}$ wherefor we write, for example, $x \not\leq_{\mathcal{L}} y$ to mean that "*x does not securely flow into y*".)

Given an arbitrary set $X \subseteq \textbf{Ide}$ that satisfies $flowOrder$, and given that $x \in X$ ranges over updated variables, we define $\forall x \in X$ a downward inclusion ordering $\downarrow X$, which collects all variables that may "*securely flow into*" x as:

$$\downarrow X \ = \ \{y \in \textbf{Ide} \mid (\exists x \in X)\ y \leq_{\mathcal{L}} x\}$$

Note: We write $\downarrow x$ to abbreviate the downward inclusion defined on the singleton set $\{x\}$, i.e. $\downarrow\{x\}$.

We also define a 'downward inclusion complement', denoted $\downarrow X^c$. In contrast to $\downarrow X$, this corresponds to some $\{z \in \mathbf{Ide} \mid z \notin \downarrow X\}$. This collects all variables $z \in \mathbf{Ide}$ mapped to security classes higher than $classify(x)$ as well as all $z \in \mathbf{Ide}$ *incomparable* with $x \in X$ by the $\leq_{\mathcal{L}}$ relation. Thus,

$$\downarrow X^c = \mathbf{Ide} \text{ - } \downarrow X$$

Using these definitions, we now present two syntactic security conditions based on our GSL abstraction $(\widehat{X}, \widehat{G}, \widehat{D})$ that, if *true*, guarantees that all substitutions are secure.

Theorem 7 (Dependency Freedom)**.** *We assert that the set of variables on which x depends, written D_x, is free of any variable(s) with a security classification higher than $classify(x)$.*

Formally, given that $X \subseteq \mathbf{Ide}$, $y \in \mathbf{ide}$ and $y \in \downarrow X^c$, then:

$$\forall x \in X \cdot (D_x \cap y = \varnothing) \qquad \text{- - } C_1$$

Note: Theorem 7 only takes into account, for all $x \in \widehat{X}$, flows collected in $\widehat{D}$. It does not deal with implicit global flows collected in $\widehat{G}$ that may affect program termination. Theorem 8 deals with that.

PROOF:
Given that $X \subseteq \mathbf{Ide}$ has the ordering $flowOrder$, we assume (by contradiction) that $\exists y \in \mathbf{Ide}$ such that $y \in \downarrow X^c$ and $y \in D_x$.

By definition of $\leq_{\mathcal{L}}$, for some $x_1 \in X$ and $x_2 \in X$ such that $(x_1, l_1) \in \mathbf{SC}$ and $(x_2, l_2) \in \mathbf{SC}$:

$$x_1 \leq_{\mathcal{L}} x_2 \text{ implies } l_1 \leq l_2, \qquad \text{hence}$$

$$x_1 \leq_{\mathcal{L}} x_2 \text{ implies } x_1 \in \downarrow x_2$$

Now, if $x_1 \in \downarrow x_2$, then $x_1 \notin \downarrow x_2^c$.

Thus, if $y \in D_x$ and $y \in \downarrow X^c$, then:

$\exists x \in X \cdot (y \not\leq_{\mathcal{L}} x)$, i.e., $classify(y) \not\leq classify(x)$ - - By definition

Hence our assumption $y \in D_x$ and $y \in \downarrow X^c$ ia a contradiction!

i.e, $\forall x \in X \cdot (D_x \cap y = \varnothing)$

This ends the proof of C_1.

Theorem 8 (Termination Freedom)**.** *Given that $\perp \in \mathcal{L}$ is the minimal element of $\mathcal{L}$, $X_\perp \subseteq$ **Ide**, and $\forall x \in X_\perp \cdot (x \mapsto \perp)$, we assert that whenever $x \in (X_\perp \cap \widehat{X})$, then no variable with a security classification higher than $\perp$, may affect termination.*

Formally, given that $X_\perp^c = \{y \in \mathbf{Ide} \mid y \notin X_\perp\}$, we have

$$\forall x \in (X_\perp \cap \widehat{X}) \cdot (X_\perp^c \cap \widehat{G} = \varnothing) \qquad \text{- - } C_2$$

Note: Since variables that may affect program termination *implicitly* flow into updated variables, Theorem 8 defines the condition that the security classification of variables that may affect termination (i.e., variables in $\widehat{G}$) must be lower than or equal to the minimal security class of updated variables, (i.e., $\perp$).

PROOF:

Assume $x \in (X_\perp \cap \widehat{X})$,

i.e., $classify(x) = \perp$ and $x \in \widehat{X}$ - - by definition

It remains to prove that $X_\perp^c \cap \widehat{G} = \varnothing$

Now, suppose (by contradiction) that $\exists y \cdot (y \in (X_\perp^c \cap \widehat{G}))$.

If $y \in X_\perp^c$, it follows that $classify(y) \not\leq \perp$, since $y \notin X_\perp$

i.e. $y \leq_{\mathcal{L}} x$

Hence if $y \in \widehat{G}$, it cannot be the case that $y \not\leq_{\mathcal{L}} x$ - - by Theorem 7. A contradiction indeed!

Hence if $x \in (X_{\perp} \cap \widehat{X})$, then

$X_{\perp}^{c} \cap \widehat{G} = \varnothing$.

And this ends the proof of C_2.

From the foregoing, for any substitution of interest to be adjudged secure, it has to satisfy both security conditions C_1 and C_2. The additional benefit of C_2 is that it ensures our information flow analysis framework is termination sensitive, i.e., it does not leak secret information *implicitly* as a result of the termination condition of the substitution.

To develop a semantic security condition that corresponds to GNI, we first specialise our characterisation of GNI introduced in Definition 2, Section 2.2.2.2 using GSL semantic notation. Recall from Section 2.2.2.2 that the GNI characterisation given basically means: *the set of observable outputs after the insertion of a secret input into an input sequence is consistent with the observable output, without the insertion of any secret input.* An alternative way of stating this is that: given any set of before values of some low security variables, any modification of the input stream by insertion of a high security variable (i.e., another set of before values with some high security variables) must yield some set of after-values that are equivalent to the after-values of the former. We write Σ_1 to denote a set of before states involving only low security variable inputs, Σ_h denotes a set of before states, involving some high security variables, $\sigma_1 \in \Sigma_1$ ranges over Σ_1, while $\sigma_h \in \Sigma_h$ ranges over Σ_h. Thus, we need to show in our semantic security condition that given a set $L \subseteq \mathbf{Ide}$ of variables,

$$\Sigma_1 \approx^{L} \Sigma_h \;\Leftrightarrow\; (\forall x \in L,\ \forall \sigma_h \in \Sigma_h,\ \exists \sigma_1 \in \Sigma_1 \cdot (\sigma_1(x) = \sigma_h(x)))$$

Theorem 9 (Generalised Noninterference for GSL)**.** *Given that variables are mapped to the members of a security lattice, $\mathcal{L} = \langle \mathcal{L}, \leq \rangle$, by the function*

classify introduced earlier and given two sets of states Σ_1, Σ_h*, with* Σ_h *involving high security variable inputs, we state that a substitution* C^ℓ *satisfies GNI if,*

$$\forall x \in \widehat{X} \wedge \forall y \in \mathbf{Ide},$$
$$\text{if } x\widehat{D}(\ell)y \text{ then } prd_{x,y}(C^\ell) \Rightarrow ((\Sigma_1 \approx^y \Sigma_h) \wedge (y \leq_{\mathcal{L}} x))$$

Theorem 9 captures the notion that whatever before-state we start from, there will always be some low security after-state that is *feasible* and whose after-values are consistent with that of any set of high security variable after-values (since the (list of) variable(s) impacting x, i.e., y is/are equivalent in all states). Hence the observer cannot deduce with certainty that a high level event has (or has not) occurred.

We provide a proof of our semantic security condition below:

PROOF:

From Theorem 4, we know that $\forall y \cdot x\widehat{D}(\ell)y,\ prd_{x,y}(C^\ell) \ \Rightarrow \ (\Sigma_1 \approx^y \Sigma_h)$

Hence, we are left to show that:

$\forall y \cdot x\widehat{D}(\ell)y,\ prd_{x,y}(C^\ell) \ \Rightarrow \ (y \leq_{\mathcal{L}} x)$

Theorem 7 requires that the set of variables on which x depends be free of any variable(s) with a security classification higher than that of x. Hence the observer cannot deduce with certainty that a high level event has (or has not) occurred. It follows therefore that $x\widehat{D}(\ell)y \ \Rightarrow \ (y \leq_{\mathcal{L}} x)$. Recall, too that $prd_{x,y}(C^\ell)$ basically relates the before values of x and y with their after values, and the existence of this relation shows that C^ℓ terminates. Hence, $\forall y \cdot x\widehat{D}(\ell)y,\ prd_{x,y}(C^\ell) \ \Rightarrow \ (y \leq_{\mathcal{L}} x)$. This ends the proof of Theorem 9.

At this juncture, we desire to show that the syntactic security conditions C_1 and C_2 imply the semantic security condition (Theorem 9).

PROOF:

Suppose C_1 and C_2 holds, it remains to prove that Theorem 9 holds, i.e.,

We need to show, given notations defined earlier, that:

(1) $\forall x \in X,\ \forall z \in \mathbf{Ide},$ if $(D_x \cap z = \varnothing)$ then

$(z \not\leq_{\mathcal{L}} x) \ \wedge \ (\Sigma_1 \approx^x \Sigma_2)$;

(2) $\forall x \in (X_\perp \cap \widehat{X}),$ if $(X_\perp^c \cap \widehat{G} = \varnothing)$ then

$\forall y \in \mathbf{Ide} \cdot (y \leq_{\mathcal{L}} x), \ \Sigma_1 \approx^y \Sigma_2$;

Part 1: We assume $\forall x \in X, \ \forall z \in \mathbf{Ide}, \ D_x \cap z = \varnothing$ holds. - - Theorem 7:

It follows therefore that: $z \in \downarrow X^c$.

It also follows from Formula 3.10 that: $z \not\leq_{\mathcal{L}} x$

Hence $\forall x \in X, \ \forall z \in \mathbf{Ide}$, if $(D_x \cap z = \varnothing)$ then $z \not\leq_{\mathcal{L}} x$

We are now left to show that:

$\forall x \in X, \ \forall z \in \mathbf{Ide}$, if $(D_x \cap z = \varnothing)$ then $\Sigma_1 \approx^x \Sigma_2$;

But if $D_x \cap z = \varnothing$, then whatever value x is mapped to, $\forall \sigma \in \Sigma$, is independent of z. Hence:

$\forall \sigma \in \Sigma_1, \ \exists \sigma_2 \in \Sigma_2 \cdot (\sigma_1(x) = \sigma_2(x))$ - - Formulae 3.5, 3.6

$\therefore \forall x \in X, \ \forall z \in \mathbf{Ide}$, if $(D_x \cap z = \varnothing)$ then $(z \not\leq_{\mathcal{L}} x) \ \wedge \ (\Sigma_1 \approx^x \Sigma_2)$.

This ends Part 1 of proof, which indicates that the syntactic security condition C_1 implies the semantic security condition articulated in Theorem 9.

Part 2: $\forall x \in (X_\perp \cap \widehat{X})$, we assume $(X_\perp^c \cap \widehat{G} = \varnothing)$ holds

We are left to show that:

$\forall y \in \mathbf{Ide} \cdot (y \leq_{\mathcal{L}} x), \ (\Sigma_1 \approx^y \Sigma_2)$ holds

Suppose $x \in \widehat{G}$, then by Theorem 5, $\Sigma_1 \approx^{\widehat{G}(\ell)} \Sigma_2$, thus $\Sigma_1 \approx^x \Sigma_2$

Recall that $X_\perp$ collects the variables mapped to the *minimal* element of the security lattice $\mathcal{L}$ - - Theorem 8

And $\forall x \in X_\perp$, if $y \leq_{\mathcal{L}} x$, it follows that $y \in X_\perp$ - - Formula 3.10

Hence $\Sigma_1 \approx^y \Sigma_2$

Also since $x \in X_\perp$, then $x \notin X_\perp^c$. (Ditto y). And since $x \in \widehat{G}$ (by our assumption), clearly: $X_\perp^c \cap \widehat{G} = \varnothing$

$\therefore$ if $(X_\perp^c \cap \widehat{G} = \varnothing)$ then

$\forall y \in \mathbf{Ide} \cdot (y \leq_{\mathcal{L}} x), \ \Sigma_1 \approx^y \Sigma_2$

This ends Part 2 of proof, which indicates that the syntactic security condition C_2 implies the semantic security condition articulated in Theorem 9.

Note: As discussed in Section 3.4.1, our information flow analysis framework collects all possible flows of information between variables in a B development. As we have just discussed here, applying both security conditions C_1 and C_2, ensures that both *explicit flows* (captured by C_1; applied to $\widehat{D}$) and termination-sensitive *implicit flows* (captured by C_2; applied to $\widehat{G}$) are prevented, and only machines with flows satisfying defined security policies are committed. The most critical *safety* concern with respect to information flow is adjudging an *insecure* program *secure*, often termed: *false negative*. Since, as noted, our $\widehat{X}$, $\widehat{G}$, $\widehat{D}$ abstraction is an over-approximation over all possible flows (Section 3.4.1), which are then checked by C_1 and C_2, it is inconceivable that any *false negative* case could arise within our analysis. Hence, on the basis of this safety condition, our information framework is okay. A somewhat less critical safety condition is adjudging a *secure* program *insecure*, termed: *false positive*. This leads to *false alarms* that may result in lost time and effort trying to fix a non-existent problem. Since both C_1 and C_2 ensure that only the values of variables with security classification *lower than or equal to* the security classification of some $x \in \widehat{X}$ may flow into x, only variables outside that set are adjudged insecure. Hence, a false positive case cannot arise.

Having discussed the relationship between the semantic and syntactic security conditions of GSL, we present in the following section a hypothetical example of how our optimised information flow analysis works.

3.7 Example Information Flow Analysis for GSL

In this section we present an example showing how our analysis framework detects flows within program constructs, and determines whether such flows are secure or insecure.

Given that z is a predicate; p, q, w, x, y are integer variables,
Ide = $\{p, q, w, x, y, z\}$, and the variables are mapped to a two-point lattice, $\langle \mathcal{L}, \leq \rangle = H \not\leq L$, as follows $f(p) = f(w) = f(x) = f(z) = L$, and $F(q) = f(y) = H$, the task is to analyse the information flow in the following program construct.

$w := 0;$
$y := 0\ ;$
while$(w < 5)$ **do**
    if$(p = q)$ **then**
        $x := y + w$
    else
        $z \Longrightarrow x := 0$
    end;
    $w := w + 1$
end;
$x := 1$

We start our analysis by labeling the various substitutions that make up our program construct. Thus the program could be re-written as follows:

$((w := 0)^{\ell_1};$
$(y := 0)^{\ell_2}\ ;$
$(\mathbf{while}(w < 5)\ \mathbf{do}$
    $((\mathbf{if}(p = q)\ \mathbf{then}$
        $(x := y + w)^{\ell_8}$
    else
        $(z \Longrightarrow x := 0)^{\ell_9}$
    $\mathbf{end})^{\ell_6};$
    $(w := w + 1)^{\ell_7})^{\ell_5}$
$\mathbf{end})^{\ell_3};$
$(x := 1)^{\ell_4})^{\ell_0}$

Our first task is to construct $\widehat{D}(\ell_0)$, using reaching dependencies analysis. The procedure for this analysis is detailed in Table 3.9.

We then construct Table 3.10 to derive $\widehat{X}$, and $\widehat{G}$ of the generalised substitutions for all $\ell \in \mathbf{Lab}$.

From the results given in Tables 3.9, and 3.10, we get the following abstractions of the example program and its sub-constructs, using the abstraction schema $(\widehat{X}, \widehat{G}, \widehat{D})$:

Blocks	Functions	Evaluation
ℓ_1	$init_{RD}(\ell_1)$	$\{(w,?),(y,?),(x,?)\}$
	$kill_{RD}(\ell_1)$	$\{(w,?)\}$
	$gen_{RD}(\ell_1)$	$\{(w,\varnothing)\}$
	$final_{RD}(\ell_1)$	$(init_{RD}(\ell_1) - kill_{RD}(\ell_1)) \cup gen_{RD}(\ell_1)$ $= \{(w,\varnothing),(y,?),(x,?)\}$
ℓ_2	$init_{RD}(\ell_2)$	$final_{RD}(\ell_1) = \{(w,\varnothing),(y,?),(x,?)\}$
	$kill_{RD}(\ell_2)$	$\{(y,?)\}$
	$gen_{RD}(\ell_2)$	$\{(y,\varnothing)\}$
	$final_{RD}(\ell_2)$	$(init_{RD}(\ell_2) - kill_{RD}(\ell_2)) \cup gen_{RD}(\ell_2)$ $= \{(w,\varnothing),(y,\varnothing),(x,?)\}$
ℓ_3	$init_{RD}(\ell_3)$	$final_{RD}(\ell_2) \cup final_{RD}(\ell_3)$ $= \{(w,\varnothing,),(y,\varnothing),(x,?)\} \cup final_{RD}(\ell_3)$
	$kill_{RD}(\ell_3)$	$kill_{RD}(\ell_5) = kill_{RD}(\ell_6) \cup kill_{RD}(\ell_7)$ $kill_{RD}(\ell_6) = kill_{RD}(\ell_8) \cup kill_{RD}(\ell_9) = \{(x,?),(z,?)\}$, and $kill_{RD}(\ell_7) = \{(w,\varnothing)\}$, i.e., $kill_{RD}(\ell_5) = \{(w,\varnothing),(x,?),(z,?)\}$
	$gen_{RD}(\ell_3)$	$gen_{RD}(\ell_5) = gen_{RD}(\ell_6) \cup gen_{RD}(\ell_7)$ $gen_{RD}(\ell_6) = gen_{RD}(\ell_8) \cup gen_{RD}(\ell_9)$ $gen_{RD}(\ell_8) = \{(x,y),(x,w),(x,p),(x,q)\}$, and $gen_{RD}(\ell_9) = \{(x,z),(x,p),(x,q)\}$, hence $gen_{RD}(\ell_6) = \{(x,w),(x,y),(x,z),(x,p),(x,q)\}$. Also, $gen_{RD}(\ell_7) = \{(w,\varnothing)\}$, therefore, $gen_{RD}(\ell_5) = \{(x,w),(x,y),(x,z),(x,p),(x,q),(w,\varnothing)\}$
	$final_{RD}(\ell_3)$	$(init_{RD}(\ell_3) - kill_{RD}(\ell_3)) \cup gen_{RD}(\ell_3)$ $= (\{(w,\varnothing),(y,\varnothing)\} - \{(w,\varnothing)\}) \cup$ $= \{(x,w),(x,y),(x,z),(x,p),(x,q),(w,\varnothing)\}$ $= \{(x,w),(x,y),(x,z),(x,p),(x,q),(w,\varnothing),$ $(y,\varnothing)\}$
ℓ_4	$init_{RD}(\ell_4)$	$final_{RD}(\ell_5) = \{(x,w),(x,y),(x,z),(x,p),(x,q),(w,\varnothing),$ $(y,\varnothing)\}$
	$kill_{RD}(\ell_4)$	$\{(x,w),(x,y),(x,z),(x,p),(x,q)\}$
	$gen_{RD}(\ell_4)$	$\{(x,\varnothing)\}$
	$final_{RD}(\ell_4)$	$(init_{RD}(\ell_4) - kill_{RD}(\ell_4)) \cup gen_{RD}(\ell_4)$ $= \{(w,\varnothing),(y,\varnothing),(x,\varnothing)\}$
ℓ_0	$init_{RD}(\ell_0)$	$init_{RD}(\ell_1) = \{(w,\varnothing),(y,\varnothing),(x,\varnothing)\}$
	$final_{RD}(\ell_0)$	$final_{RD}(\ell_4) = \{(w,\varnothing),(y,\varnothing),(x,\varnothing)\}$

Table 3.9: Analysis: Desk-checking Reaching Dependencies

Lab	$\widehat{X}$	$\widehat{G}$
ℓ_1	$\{w\}$	$\varnothing$
ℓ_2	$\{y\}$	$\varnothing$
ℓ_3	$\{x,w\}$	$\widehat{G}(\ell_5) \cup FV(w<5) = (\widehat{G}(\ell_6) \cup \widehat{G}(\ell_7); \widehat{D}(\ell_6)) \cup FV(w<5)$ $= ((\widehat{G}(\ell_8) \cup \widehat{G}(\ell_9)) \ \cup \ (\widehat{G}(\ell_7); \widehat{D}(\ell_6)) \cup FV(w<5)$ $= \varnothing \cup \varnothing \cup \varnothing \cup \{w\} = \{w\}$
ℓ_4	$\{x\}$	$\varnothing$
ℓ_5	$\{w,x\}$	$\widehat{G}(\ell_6) \cup \widehat{G}(\ell_7); \widehat{D}(\ell_6)$ $= (\widehat{G}(\ell_8) \cup \widehat{G}(\ell_9)) \ \cup \ \widehat{G}(\ell_7); \widehat{D}(\ell_6)$ $= \varnothing \cup \varnothing \cup \varnothing \ = \ \varnothing$
ℓ_6	$\{x\}$	$\varnothing$
ℓ_7	$\{w\}$	$\varnothing$
ℓ_8	$\{x\}$	$\varnothing$
ℓ_9	$\{x\}$	$\varnothing$
ℓ_0	$\{w,x,y\}$	$\widehat{G}(\ell_3) \ \cup \ \widehat{G}(\ell_4); \widehat{D}(\ell_3) = \{w\} \ \cup \ \varnothing = \{w\}$

Table 3.10: GSL Abstraction of Example Flow Analysis ($\widehat{G}$)

$$FA(C^\ell) \xrightarrow{\alpha} (\widehat{X}, \widehat{G}, \widehat{D})$$

We have:

$$FA(\ell_1) \triangleq (\{w\}, \varnothing, \{(w,\varnothing),(y,?),(x,?)\})$$
$$FA(\ell_2) \triangleq (\{y\}, \varnothing, \{(w,\varnothing),(y,\varnothing),(x,?)\})$$
$$FA(\ell_3) \triangleq (\{w,x\}, \{w\}, \{(x,w),(x,y),(x,z),(x,p),(x,q),(w,\varnothing),(y,\varnothing)\})$$
$$FA(\ell_4) \triangleq (\{x\}, \varnothing, \{(w,\varnothing),(y,\varnothing),(x,\varnothing)\})$$
$$FA(\ell_5) \triangleq (\{w,x\}, \varnothing, \{(x,w),(x,y),(x,z),(x,p),(x,q),(w,\varnothing),(y,\varnothing)\}\})$$
$$FA(\ell_6) \triangleq (\{x\}, \varnothing, \{(x,w),(x,y),(x,z),(x,p),(x,q)\})$$
$$FA(\ell_7) \triangleq (\{w\}, \varnothing, \{(w,\varnothing)\})$$
$$FA(\ell_8) \triangleq (\{x\}, \varnothing, \{(x,w),(x,y)\})$$
$$FA(\ell_9) \triangleq (\{x\}, \varnothing, \{(x,z)\})$$
$$FA(\ell_0) \triangleq (\{w,x,y\}, \{w\}, \{(w,\varnothing),(y,\varnothing),(x,\varnothing)\})$$

The last stage of our flow analysis is to check whether ℓ_0 satisfies the security conditions C_1 and C_2 (*in which case the program is secure*) or not (*in which case the program is insecure*). From the given security classification of the variables, we can deduce the following with respect to $\widehat{X}(\ell_0)$:

Recall that $\widehat{X}(\ell_0) = \{w,x,y\}$, $f(y) = H$ and $f(w) = f(x) = L$. Hence

Low security variables :: $\downarrow\{w,x\} \subset \widehat{X}(\ell_0) = \{w,x\}$, and
High security variables :: $\downarrow\{w,x\}^c = \{y\}$
Checking C_1 with respect to $\{a \in \widehat{X}(\ell_0) \mid a \in classify^{-1}(L)\}$:

$$\forall a \in \{w,x\} \cdot (D_a \cap \downarrow\{w,x\}^c) = \{w,x\} \cap \{y\} = \varnothing$$

Therefore C_1 holds for all $a \in \{w,x\}$.

Similarly,

$\downarrow\{y\} \subset \widehat{X}(\ell_0) = \{w,x,y\}$, and
$\downarrow\{y\}^c = \varnothing$

Checking C_1 with respect to $\{b \in \widehat{X}(\ell_0) \mid b \in classify^{-1}(H)\}$:

$$\forall b \in \{y\} \cdot (D_b \cap \downarrow\{y\}^c) = \{y\} \cap \varnothing = \varnothing$$

Therefore C_1 holds for all $b \in \{y\}$.

Conclusion: Since $\{w,x\} \cup \{y\} = \widehat{X}(\ell_0)$, and C_1 holds for all $a \in \{w,x\}$, and C_1 also holds for all $b \in \{y\}$, it follows that C_1 holds for all variables in $\widehat{X}(\ell_0)$.

Again, from the security classification, with respect to $\widehat{X}(\ell_0)$, $X_\perp = \{w,x\}$, and $X_\perp^c = \{y\}$. Thus,
Checking C_2:

$$\forall a \in X_\perp \cdot (X_\perp^c \cap \widehat{G}) = \{y\} \cap \{w\} = \varnothing$$

Conclusion: C_2 holds.

The results therefore show that the program block labeled ℓ_0 is secure since both C_1 and C_2 holds.

Discussion:
Our analysis is flow-sensitive and more robust than type-based frameworks like the Volpano and Smith [130], [76] type system in the literature. Most type-based systems will reject the program block labeled ℓ_0 as *insecure*, whereas with the notion of security under consideration being that the '*initial value of high security variables may not interfere with the final value of low-security variables*' (i.e., no intermediate reading of variables is allowed), ℓ_0 is clearly secure although some of its subconstructs are not secure.

For example, notice from our application of the security conditions to ℓ_6 below that ℓ_6 on its own is insecure.

$$\widehat{X}(\ell_6) \;=\; \{x\}$$
$$\downarrow\widehat{X}(\ell_6) \subset \mathbf{Ide} = \{x, w, z, p\},$$
$$\downarrow\widehat{X}(\ell_6)^c = \{y, q\}, \qquad \text{and}$$
$$\forall a \in \widehat{X}(\ell_6),\ D_a = \{y, w, z, p, q\}$$

Checking C_1:

$$D_a \cap \downarrow\widehat{X}(\ell_6)^c \;=\; \{y, w, z, p, q\} \cap \{y, q\} \;=\; \{y, q\}.$$
$$\text{i.e.,}\quad D_a \cap \downarrow\widehat{X}(\ell_6)^c \;\neq\; \varnothing$$

Conclusion: ℓ_6 fails C_1.

Since both security conditions *must* be satisfied for secure information flow to be guaranteed, there is no need to check for C_2. Thus, we conclude that ℓ_6 is insecure since it fails C_1.

When ℓ_6 is embedded within the program block ℓ_0, as in our example, however, the *insecurity* implicit in ℓ_6 does not impact on what an adversary is able to learn about the initial value of any high security variable after multiple runs of the program. The reason for this is that the alternation condition $p = q$ does not propagate to the output, i.e., an adversary cannot deduce which branch of the conditional is taken, since ℓ_4 overwrites whatever *insecure* update ℓ_6 might have made to x (thus for all runs of ℓ_0, the final value of x is always 1). Hence an adversary is unable to infer either that $p = q$ or $p \neq q$, or whether y flows into x or not when ℓ_6 is embedded within ℓ_0.

Having illustrated with an example how information flow is analysed using our reaching dependencies framework, we now follow in Section 3.8 with a generic distributive analysis framework.

3.8 Analysing Information Flow for GSL using a Monotone Framework

The reaching dependencies analysis introduced in Section 3.5 can be represented using a generic framework that is unchanging whatever the anal-

ysis type used. This framework is referred to as *monotone framework* in the literature (see Hankin and the Nielsons, [64]). The benefit of such a framework is that it enables the development of generic algorithms independent of the information flow analysis method used, thereby making the algorithms more portable. The monotone framework introduced here is designed to handle *forward analysis*[7], but it can easily be adapted to deal with its dual, *backward analysis*,[8] as well. We state below the notation used in our monotone framework.

L denotes the complete lattice spanned by the information flow property space, and is defined as $L \in \mathcal{P}(\mathbf{Ide} \times \mathbf{Ide})$. This property space defines, for each label $\ell \in \mathbf{Lab}$: the *finite* set of all dependency pairs, i.e. $\mathcal{P}(\widehat{X} \times (D_x \cup \widehat{G}))$;

E collects *extremal* labels, i.e., initial and final sub-labels in a composite program. Hence, given that S denotes a GSL substitution, E is either $init(S)$ or $final(S)$; $\bigsqcup$ means $\bigcup$, $\sqcup$ is $\cup$ and $\sqsubseteq$ means $\leq$;

$\iota \in L$ is an *extremal* value for the extremal label $\ell \in E$;

F collects the flow of information between two labeled semantic program blocks, i.e., given that S denotes a GSL substitution with two or more semantic program blocks, then F is $flow(S)$;

f_ℓ collects the *transfer functions*[9] associated with semantic program blocks, $B^\ell \in blocks(S)$, in GSL substitution S, i.e., $f_\ell \in L \to L$;

$\mathcal{F}$ denotes the set of all monotone functions of interest over L, the identity function inclusive;

f denotes a mapping from the labels **Lab** of F and E to dependency pairs, i.e., transfer functions in $\mathcal{F}$, i.e., $f \in \mathbf{Lab} \to \widehat{D}$;

$Analysis_o$ defines entry conditions to a semantic block, while

$Analysis_\bullet$ defines exit conditions to a semantic block.

The monotone framework gets its name from the requirement that the transfer functions are monotone, i.e., given $l_1, l_2 \in L$, ordered by $\langle L, \sqsubseteq \rangle$, then $l_1 \sqsubseteq l_2 \Rightarrow f_\ell(l_1) \sqsubseteq f_\ell(l_2)$. The $\sqsubseteq$ ordering describes the property that the *lhs* precedes the *rhs* (when used as an infix operator as in $l_1 \sqsubseteq l_2$) in L. We have

[7] The properties of program inputs are used to determine the properties of outputs.

[8] The properties of program inputs are determined using the properties of outputs.

[9] In the context of our work, *transfer functions* capture the cumulative information flow (dependencies) at each semantic program block.

the choice of (1) developing either the general iterative algorithm for Monotone Frameworks, termed *Maximal Fixed Point* (MFP), which computes the least fixed point of our flow analysis, or (2) using a *Meet Over all Paths* (MOP) solution. A MOP solution directly propagates analysis information from entry point along all possible program paths up to the program point of interest. We have chosen to use the MFP approach because the property space satisfies the *Ascending Chain Condition*[10], making the MFP always computable, whereas the MOP solution does not satisfy this condition, since generally, there are infinite number of possible paths to a program point, hence it (MOP) is *undecidable* [64].

A MFP instance of a monotone framework is defined with the schema:

$$MFP \triangleq (L, \mathcal{F}, F, E, \iota, f)$$

Before we can develop a MFP information flow analysis, however, we need definitions for both $Analysis_o$ and $Analysis_\bullet$, and these we present below:

We define the *unit* of our property set with respect to the least upper bound, $\sqcup$, as $\bot$. This effectively is the least element of the ordered set $\langle L, \sqsubseteq \rangle$. Thus for any $l \in L,\ l \sqcup \bot = l$, i.e., $\bot$ corresponds to the identity for $\sqcup$.

Given that:

$$\iota^{\ell}_{E} = \begin{cases} \iota & \text{if } \ell \in E, \\ \bot & \text{if } \ell \notin E. \end{cases}$$

We have

$$Analysis_o(\ell) = \bigsqcup\{Analysis_\bullet(\ell') | (\ell', \ell) \in F\} \sqcup \iota^{\ell}_{E}$$

and

$$Analysis_\bullet(\ell) = f_\ell(Analysis_o(\ell))$$

From the foregoing, it follows that if $\ell \notin E$, then the extremal value, $\iota^{\ell}_{E} = \bot$. Intuitively, this indicates that there is no extremal value to add into the

[10] Defined on a partially ordered set, the ascending chain condition requires that all increasing sequences of the set eventually become constant, i.e, in $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$, $\exists n \cdot (a_n = a_{n+1} = a_{n+2} = \ldots)$.

flow analysis, so the analysis defaults to the cumulative analysis up to that program point, as shown formally below:

$$\begin{aligned} Analysis_o(\ell) &= \bigsqcup\{Analysis_\bullet(\ell') | (\ell', \ell) \in F\} \sqcup \bot \\ &= \bigsqcup\{Analysis_\bullet(\ell') | (\ell', \ell) \in F\} \end{aligned}$$

Information flow analysis using a monotone framework requires a *worklist algorithm*. A worklist, denoted W, is a list of pairs of the elements of the flow relation F, which indicates a change to the analysis on exit from the block labeled by the first element of the pair, and hence need to be recalculated on entry to the block labeled by the second element of the worklist pair [64]. Given that $W = [(\ell_1, \ell_2), (\ell_2, \ell_3), \ldots (\ell_{n-1}, \ell_n)]$, we define the head of the list in psuedocode as $head(W) = (\ell_1, \ell_2)$ and the tail, denoted $tail(W)$, means $W - head(W)$. We write $fst(head(W))$ to refer to the first element of the pair in the head, i.e., ℓ_1; and $snd(head(W))$ refers to the second element in the head, i.e., ℓ_2. Adopting a c++ vector[11] data structure, we write $vect(W)$ to denote the store for the worklist elements. We also assume that, like c++ vectors, $vect(W)$ has no predetermined size, but that it can be expanded to include any number of pairs of worklist elements, depending on the number of semantic blocks in the substitution. We assume a built-in method for updating elements in a worklist, again like in C++, this is written for $vect(W)$ for example, as $vect.pushback(\ldots)$, where '$\ldots$' denotes the element to be added to the vector. The basic intuition behind the worklist algorithm is to iterate through the list, starting from the start (*extremal*) label and capturing the dependency pairs cumulatively until the *least fixed point* is reached in the dependency collections.

Now, given that $S \triangleq C^\ell$ is a labeled GSL substitution, we present, in pseudocode below, an algorithm to solve the information flow equations for S. Notice that we present the results in a form similar to the flow analysis, thus MFP_o denotes the MFP for the entry to a semantic program block, whereas $MFP_\bullet$ denotes the MFP for the exit from a semantic program block.

[11] Like arrays, a vector is a container that holds a strictly linear sequence of elements indexed in much the same way as arrays. The main advantage of vectors over arrays is that the former is elastic - it can be expanded and contracted as necessary, whereas the latter is rigid.

Worklist Algorithm for solving Information Flow Equations

INPUT: An instance of a monotone framework, i.e., $(L, \mathcal{F}, F, E, \iota, f)$

OUTPUT: MFP_o, $MFP_\bullet$

METHOD: Step 1: Initialisation of W and $Analysis$

$W := \text{nil};$
for all $(\ell, \ell') \cdot flow(C^{\ell'})$ in F do
$W := vect((\ell, \ell'), W);$
$i.e., W := vect.pushback((\ell, \ell');$ //builds vector
for all ℓ in E or $\ell \cdot flow(C^{\ell})$ in F do
if $\ell \in E$ then $Analysis[\ell] := \iota$
else $Analysis[\ell] := \bot;$

Step 2: Iteration (updating W and $Analysis$)

while $W \neq$ nil do
$\ell := fst(head(W));\ \ell' := snd(head(W));$
$W := tail(W);$
if $f_\ell(Analysis[\ell] \notin Analysis[\ell']$ then
// '$\sqcup$' implicitly removes duplicate dependencies
$Analysis[\ell'] := Analysis[\ell'] \sqcup f_\ell(Analysis[\ell]);$
for all ℓ'' with $(\ell', \ell'') \in F$ do
$W := vect((\ell', \ell''), W);$

Step 3: Presenting the result (MFP_o and $MFP_\bullet$)

for all ℓ in E or $\ell \cdot\ flow(C^{\ell})$ in F do
$MFP_o(\ell) := Analysis[\ell];$
$MFP_\bullet(\ell) := f_\ell(Analysis[\ell])$

Using the monotone framework described in this section, we can solve the set of information flow equations derived from the example program discussed earlier in Section 3.7. The result, presented in Table 3.11, shows the same conclusion we derived in Section 3.7 (where reaching dependencies analysis was used), that the program is secure. For convenience, we re-present the example here, with a slight change to the labeling, and in Table 3.11, we write '?' to denote undefined value. Notice that iterations 8 and 11 in Table 3.11 show that our analysis satisfies the ascending chain condition.

$$
\begin{array}{l}
[\ [w := 0]^{\ell_1}; \\
[y := 0]^{\ell_2}\ ; \\
[\textbf{while}(w < 5)\ \textbf{do} \\
\quad [\textbf{if}(p = q)\ \textbf{then} \\
\quad\quad [x := y + w]^{\ell_3} \\
\quad \textbf{else} \\
\quad\quad [z \Longrightarrow x := 0]^{\ell_4} \\
\quad \textbf{end}]^{\ell_5}; \\
\quad [w := w + 1]^{\ell_6} \\
\textbf{end}]^{\ell_7}; \\
[x := 1]^{\ell_8}\ ]^{\ell_0}
\end{array}
$$

3.9 Analysing the Computational Complexity of GSL Flow Analysis

In this section we analyse the efficiency and scalability of our GSL information flow analysis framework in terms of notions of computational *complexity*, which, counter-intuitively, have no bearing on how complicated (or *complex*) a computational process is, but rather means '*the amount of work done*' (or amount of resources used) by a computational process or algorithm [15] [44]. This notion of '*amount of work done*' can be measured in terms of speed, memory usage, transmission size (e.g. bandwidth), etc. The most commonly used complexity measure is the *relative speed* of an algorithm. This is the measure of the order of growth of the running time of an algorithm relative to variations in the input size (i.e., performance changes as the size of input data set increases). Thus, the complexity measure we employ in this thesis is relative speed. We do not lose any generality by doing this, because the same principles employed in the order of growth of running time can be easily modified for order of growth of memory (or bandwidth) usage, or other metrics [44]. We will also be analysing the '*rate of growth*' relative to increased data size to determine the scalability of our information flow analysis.

The metrics that we used to capture the precise time for our GSL flow analysis to complete its tasks include input size, number of basic operations based on particular inputs, number of steps within operations, etc. To cal-

Flow ANALYSIS USING MONOTONE : FRAMEWORK										
Iter	**Analysis**$[\ell]$ *for* ℓ *i.e.,* $FA(C^{\ell})$									
	W	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7	ℓ_8	ℓ_0
1	nil	$\varnothing$	?	?	?	?	?	?	?	?
2	$((\ell_1,\ell_2),W)$	$\varnothing$	$\varnothing$	?	?	?	?	?	?	?
3	$((\ell_2,\ell_3),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	?	?	?	?	?	?
4	$((\ell_2,\ell_4),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	?	?	?	?	?
5	$((\ell_2,\ell_5),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	?	?	?	?
6	$((\ell_5,\ell_6),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	?	?	?
7	$((\ell_2,\ell_7),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	?	?
8*	$((\ell_7,\ell_2),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	?	?
9	$((\ell_7,\ell_8),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$\varnothing$	?
10	$((\ell_8,\ell_0),W)$	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$\varnothing$	$\varnothing$
11*	( )	$\varnothing$	$\varnothing$	$[(x,y),(x,w)]$	$\varnothing$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$[(x,y),(x,w)]$	$\varnothing$	$\varnothing$

Table 3.11: Information Flow Analysis using Monotone Framework

culate the computational complexity of our GSL flow analysis, we seek to capture the '*worst-case*' time complexity. Once this is done, we can then confidently conclude that, based on a series of large input values, it is not the case that any instance of our GSL flow analysis, $f(n)$, will perform any worse than the worst-case instance, $g(n)$. Hence, since for all possible inputs, the computational complexity is always *less than or equal to* (i.e., 'tends to' or 'approaches') the worst-case complexity, the worst-case complexity constitutes an *asymptotic upper bound* to the function on all possible input values. The notation in the literature for capturing this asymptotic behaviour of functions, attributed to P. Bachmann in his book *Analytische*

Zahlentheorie ("Analytical Number Theory") in 1892, is termed the ' Big-O' Notation [122] [139]. We present below the formal definition of the Big-O notation.

Definition 14 (Big-O)**.** *Given that n is the size of the input; c, k are positive integer constants. Let $f(n)$ and $g(n)$ denote two distinct asymptotically non-negative functions*[12]*. Then $f(n) \in O(g(n))$ is defined as ...*

$$O(g(n)) = \{f(n) \mid (\forall n \geq k \wedge c > 0 \wedge k > 0)\ 0 \leq f(n) \leq c.g(n)\}$$

It is worth noting that although the Big-O notation is often written either as $f(n) = O(g(n))$, or $f(n) \in O(g(n))$, the notion does not mean either equality of values or set membership of values as the notation appear to suggest. An examination of the formal definition given above shows that what we have is more appropriately described as an approximation of a set of functions, and not a set of values. The Big-O notation could be graphically depicted as shown in Figure 3.5.

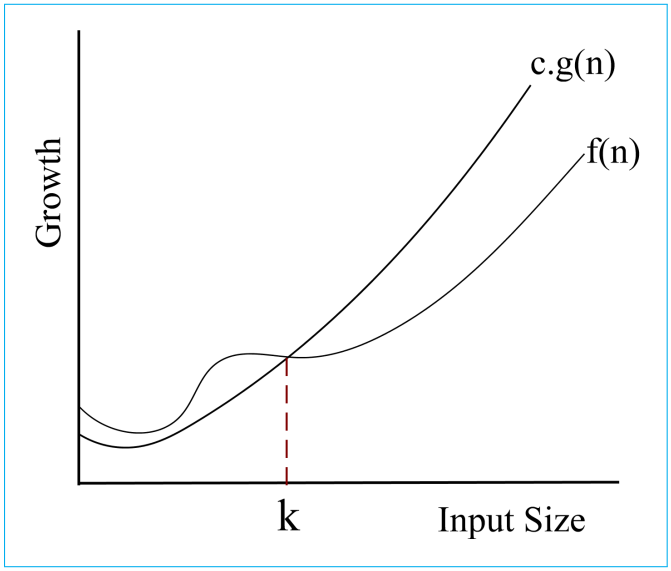


Figure 3.5: Big O Growth Graph

[12] Asymptotically non-negative functions are functions that are non-negative for all sufficiently large input, n. We consider only $\lfloor f(n) \rfloor$ here, i.e., $f(n)$ rounded down to the *nearest lower integer* value.

Before we proceed with the measurement of the efficiency and scalability of our GSL flow analysis, we discuss the model of computation used for our complexity analysis. This is the subject of Section 3.9.1.

3.9.1 The Random Access Machine (RAM) Model

Recall that our objective is to measure the 'order of growth' and the 'rate of growth' of our GSL flow analysis. Hence our primary concern with the choice of the model of computation used in the complexity analysis is simply that the estimated amount of work done be *roughly* proportional to the actual amount of work done in the GSL flow analysis. To ensure the generality of our complexity analysis, we also desire that the model be independent of hardware, operating system, or programming language used. Thus, we employ an abstract model of computation, somewhat similar to the Turing machine model. This model is termed the '*Random Access Machine (RAM) Model*'. The RAM is a generic *uni-processor* machine on which instructions can only be executed sequentially (not concurrently) [44] [15]. Another component of the RAM is an unbounded bank of numbered memory cells that can store some arbitrary data. Retrieving data from any memory cell is assumed to take one unit of time. Meulen et al in [105] noted that the processor here can be likened to the Turing machine's transition function and register, whereas the bank of memory cells is akin to the 'scratch pad' in Turing machines.

The RAM is an input-output processing system in that it takes an input (usually a large input), processes the instructions defined by the algorithm under analysis, and outputs the cost of the analysis. While it is important to capture the amount of work done as precisely as necessary, too much precision will bog down our analysis of the costs of the computation. At the other extreme, we do not want to be too general either, as the resulting costs would not be proportional to the actual processing done by the algorithm [4] [85], [135]. Without loss of generality, we assume that our RAM model takes as input a semantic program block, which is modified by increasing the number of primitive instructions involved in the block, and outputs an integer expression (or value) corresponding to the time cost of the particular GSL flow analysis under consideration. For ease of comparison, we consider the input size to be the number of primitive instructions in the semantic

block under consideration. For example, for a simple substitution, the input size is the number of variables on the right hand side of an assignment (since we need to capture these variables in our analysis), whereas for a *while* GSL construct, the input size is the number of iterations until the program terminates. We are now ready to introduce the instructions required to execute our GSL flow analysis on a RAM, but first, we present below the notations used here, viz:

- Recall that we write $\widehat{X}$ for variables assigned to, and D_x denotes the set of variables on which a variable $x \in \widehat{X}$ depends. Here, we introduce D_x_Array, which as the name implies, denotes an array in RAM storing all variables in D_x. Thus, given that i is a non-negative integer variable, we write $D_x\_Array[i]$ to denote the variable stored at the $(i+1)^{th}$ position in the array D_x_Array.
- Recall, too, that we write $FV(E)$ to denote the free variables in the expression E. In addition to that we here use $FV(E)_0$ to denote the *first* free variable in E, i.e., the variable at the *front* of the list
- We write $parse(E)$ to denote a command to read the expression E
- Recall also that we write $\widehat{D}, \widehat{G}$ to denote the set of pairs of variables, dependencies, and the set of variables that may affect termination, '*Globals*' respectively. We write $\widehat{D}(\ell)\_Array$ to denote an array of pairs of dependencies, whereas, we write $\widehat{G}(\ell)\_Array$ for the array of the variables that may affect termination.

With the notation given above we now re-present our information flow analysis, given earlier in Table 3.4b, as a class of RAM algorithms. We then use this to compute the time complexity of our analysis of the example.

Simple Substitution $(x := E)^{\ell}$:
Recall that $\widehat{D}(\ell) \supseteq Id[x \mapsto FV(E)]$

The work done to capture the dependencies for this simple substitution in our GSL flow analysis framework is given by the following semi-formal algorithm.

Input: $(x := E)^{\ell}$, where input size $= n$
Output: the total number of instructions in $(x := E)^{\ell}$
Initialisation: $i := 0,\ D_x\_Array := \{\}\ output := \{\}$
The RAM Algorithm:

$parse(E)$ - - 1
$while(FV(E) \neq \varnothing)$ - - 2
do
  $begin$
    $D_x\_Array[i] := FV(E)_0$ - - 3
    $output := output\ \cup\ \{(x, D_x\_Array[i])\}$ - - 4
    $FV(E) := FV(E) - FV(E)_0$ - - 5
    i^{++} - - 6
  end
$return\ output$ - - 7

To compute the complexity for the simple substitution algorithm, we attribute 1 cost unit to each instruction in the algorithm. Thus, we detail in Table 3.12 how we arrive at the total number of instructions in our GSL flow analysis of $(x := E)^{\ell}$ by a summation of the individual instructions within the algorithm.

From Table 3.12, we compute the total running time of our GSL flow analysis of $(x := E)^{\ell}$ to be $5n + 3$, i.e., $f(n) = 5n + 3$. Now assume this yields a complexity of the order of n, i.e., $5n + 3 \in O(n)$. For this to be correct, there must be some constant, c where $c > 0$, and some *integer* constant, k, where $k > 0$ and $n \geq k$ such that $0 \leq 5n + 3 \leq c.g(n)$ (from Definition 14).
Now, take $c = 9$, and $k = 1$, then when $n = k$, we have:

$$0 \ \leq\ 5(1) + 3 \ \leq\ 9.g(1) \qquad \text{–by substitution}$$

$$\text{i.e., } 0 \ \leq\ 8 \ \leq 9g(1)$$

Recall that $g(n)$ is asymptotically non-negative. Hence $g(1) \geq 1$, and by implication, $9g(1) \geq 9$. Thus, $0 \ \leq\ 8 \ \leq 9g(1)$ evaluates to TRUE. We can therefore conclude that $5n + 3 \in O(n)$, *for all* $n \geq 9$. The factor, 5, and the constant, 3, do not depend on n, hence the time complexity of our GSL flow

OVERALL RUNNING TIME : $(\mathbf{x} := \mathbf{E})^{\ell}$		
Instructions	**Running Time**	**Comments**
1	1	Executes only once
2	$n + 1$	Loop condition executed until false, i.e., once more than loop body
3	n	Executes n times
4	n	Executes n times
5	n	Executes n times
6	n	Executes n times
7	1	Executes only once
Total running time = 5n + 3		

Table 3.12: Computing overall running time of GSL Analysis of $(x := E)^{\ell}$

analysis for simple substitutions is $O(n)$.

Observation.
We observe from our complexity analysis above that any value of c greater than or equal to 8, and any value of k greater than 0 will also satisfy the inequality $0 \leq 5n + 3 \leq c.g(n)$. Also, the coefficient, 5, of the variable n and the constant, 3, does not impact the complexity of the analysis, hence the reason why we assumed the initialisations were done outside the *generic* algorithm for simple substitutions, because even if they were done within, the complexity would still remain unaffected. From this observation, we can safely conclude that the complexity for multiple substitutions will also be $O(n)$ since $O(n) + O(n)... = O(n)$. Note that for the semantic program block, *skip*, we do not need to compute any complexity, since there is 'no work done'.

Before considering other GSL semantic program blocks, we first consider

the time complexity of our GSL flow analysis on iterations, since iterations sometimes occur within some of the other program blocks. We use the following generic algorithm for iterations defined in semi-formal pseudocode.

Iteration (WHILE b **DO** $C_1^{\ell_1}$ $INVARIANT$ J $VARIANT$ V **END**$)^{\ell}$:

Recall that $\widehat{G}(\ell) \supseteq FV(b) \cup \widehat{G}(\ell_1) \cup \widehat{G}(\ell); \widehat{D}(\ell_1)$, and
$\widehat{D}(\ell) \supseteq \widehat{D}(\ell); \widehat{D}(\ell_1) \cup (\widehat{X}(\ell) \times FV(b))$

Hence the work done to capture the dependencies and termination behaviour of iterations in our GSL flow analysis is shown by the following semi-formal algorithm. First, we assume the body of the *loop* contains a simple substitution, and then consider the case when the loop itself contains an internal loop. To indicate that the variant decrements as required, we write v^{--} in the body of the loop as shown below.

$\widehat{D}(\ell)\_Array$ denotes the container, an array, where all dependency pairs are stored, while
$\widehat{G}(\ell)\_Array$ denotes the container where all variables that may affect program termination are stored.

Input: Assume input size of $x := E$, i.e., number of variables in $FV(E)$, is n, and m denotes input size of loop, i.e., number of iterations. (Notice here that we write $\widehat{D}(\ell_1)\_Out$ to denote the output of $(x := E)^{\ell_1}$ and *loopOut* denotes the output of the *while* GSL)
Output: The total running time cost of loop, denoted *loopOut*.
Initialisation: We assume initialisation of all variables
The RAM Algorithm:

while b
do
begin
$(x := E)^{\ell_1}$
v^{--}
$\widehat{D}(\ell)\_Array := \widehat{D}(\ell)\_Array \ ; \widehat{D}(\ell_1)\_Out$
$\widehat{G}(\ell)\_Array := \widehat{G}(\ell)\_Array \ ; \widehat{D}(\ell_1)\_Out$
end

$\widehat{D}(\ell)\_Array := \widehat{D}(\ell)\_Array \cup (\widehat{X}(\ell) \times FV(b))$
$\widehat{G}(\ell)\_Array := \widehat{G}(\ell)\_Array \cup (FV(b) \cup \widehat{G}(\ell_1))$
return *loopOut*

This general computational analysis of work done simply indicates:

(1) The work done to iteratively capture the $\widehat{D}(\ell)\_Array$ from $FV(E)$, and thereafter append the result of $\widehat{X}(\ell) \times FV(b)$ to $\widehat{D}(\ell)\_Array$, and

(2) The work done to capture $\widehat{G}(\ell)\_Array$ for the *while* GSL.

However, this RAM analysis is too general to relate sufficiently to the work done in our information flow analysis of $(x := E)^{\ell_1}$. To be able to fully capture the time complexity of this algorithm, we need to expand the work done on $(x := E)^{\ell_1}$ as shown below (Note that we use $FV(E)_0$ to denote the first (or front) variable in $FV(E)$ at each pass of our analysis over E. $FV(E)_0$ is then appended to an array, $D_x\_Array[i]$, before being removed from $FV(E)$ by the task $FV(E) := FV(E) - FV(E)_0$):

The Expanded RAM Algorithm:

while b - - 1
do
begin
while$(FV(E) \neq \varnothing)$ - - 2
do
begin
$D_x\_Array[i] := FV(E)_0$ - - (i)
$\widehat{D}(\ell_1)\_Out := \widehat{D}(\ell_1)\_Out \cup \{(x, D_x\_Array[i])\}$ (ii)
$FV(E) := FV(E) - FV(E)_0$ - - (iii)
i^{++} - - (iv)
end
return $\widehat{D}(\ell_1)\_Out$ - - (v)

v^{--} - - (a)
$\widehat{D}(\ell)\_Array := \widehat{D}(\ell)\_Array \,;\, \widehat{D}(\ell_1)\_Out$ - - (b)
$\widehat{G}(\ell)\_Array := \widehat{G}(\ell)\_Array \,;\, \widehat{D}(\ell_1)\_Out$ - - (c)
end

$\widehat{D}(\ell)\_Array := \widehat{D}(\ell)\_Array \cup (\widehat{X}(\ell) \times FV(b))$ - - 3

$\widehat{G}(\ell)\_Array := \widehat{G}(\ell)\_Array \cup (FV(b) \cup \widehat{G}(\ell_1))$ - - 4

return *loopOut* - - 5

Assigning 1 cost unit to each instruction in the algorithm, we compute the time complexity of our GSL flow analysis of a generic *while* loop and tabulate the results in Table 3.13. It is worth pointing out, however, that given that $0 < i \leq m$, for every iteration i of the while loop, C^{ℓ}, there are n iterations of the inner loop that capture $FV(E)$. Thus, for m iterations of C^{ℓ}, we have altogether $m.n$ units of work done.

We now consider the time complexity of our GSL flow analysis of alternations using the following generic algorithm for alternations defined in semi-formal pseudocode. Here we assume S_1 is an r-deep *while* loop, where r range over natural numbers, whereas S_2 is any substitution with less number of embedded loops (if any) than S_1. Hence the worst-case time complexity for the alternation corresponds to the time complexity of S_1. So, to simplify the analysis, we only need to compute the time complexity of analysis of S_1. From Table 3.13, the time complexity of a single iteration is $m.n$. Hence, without loss of generality, we assume the time complexity of r-iterations within S_1, denoted S_{1RAM}, is $(m.n)^r$. We use b_Array to denote a collection of free variables ($FV(b)$) in the alternation's condition, b; while $FV(b)_0$ denotes the front element in $FV(b)$ at each parse of the free variables.

Alternation (IF b THEN S_1 ELSE S_2 END)$^{\ell}$:

Input: **(IF** b **THEN** S_1 **ELSE** S_2 **END)**$^{\ell}$:
Output: The total running time cost of alternation, denoted *altOut*.
Initialisation: $i := 0,\ b\_Array := \{\},\ output := \{\}$
The RAM Algorithm: (assume $p : NAT$ = number of free variables in b)

parse(b)
while ($FV(b) \neq \varnothing$) - - 1
do
begin
$b\_Array[i] := FV(b)_0$ - - (i)
$FV(b)\_out := FV(b)\_out \cup \{(x, b\_Array[i])\}$ - - (ii)

OVERALL RUNNING TIME : Iterations		
Instructions	**Running Time**	**Comments**
1	$m + 1$	loop condition executed until false, i.e., once more than loop body
2	$n + 1$	Loop condition executed until false, i.e., once more than loop body
(i)	$m.n$	Executes $m.n$ times
(ii)	$m.n$	Executes $m.n$ times
(iii)	$m.n$	Executes $m.n$ times
(iV)	$m.n$	Executes $m.n$ times
(v)	m	Executes m times
(a)	m	Executes m times
(b)	m	Executes m times
(c)	m	Executes m times
3	1	Executes only once
4	1	Executes only once
5	1	Executes only once
Total running time = **4m.n + 5m + n + 5**		

Table 3.13: Computing overall running time of GSL Analysis of *Iterations*

$FV(b) := FV(b) - FV(b)_0$ - - (iii)
i^{++} - - (iv)
end
return $FV(b)\_out$ - - (v)
$parse(S_1)$
$S_{1RAM} = (m.n)^r$ - - 2
return $altOut$ - - 3

Assigning 1 cost unit to each instruction in the algorithm, we compute the time complexity of our GSL flow analysis of an alternation with an r-deep embedded *while* loop and tabulate the results in Table 3.14. The estimated total running time is $(m.n)^r + 5p + 3$, hence the time complexity of alternation corresponds to the term with the largest growth, i.e., $(m.n)^r$. We can safely conclude therefore that $(m.n)^r + 5p + 3 = O(\ (m.n)^r\ )$. This result shows that the higher the depth of *while* loops embedded within an alternation, the slower our analysis performs in computing the information flow between variables in the substitution. We note, however, that most algorithms fare poorly in the presence of deeply-embedded recurring iterations so it is generally not good programming practice to write programs that behave like that unless the very nature of the problem makes it imperative.

NOTE: Since B GSL sequential substitutions, $S_1; S_2$, may also contain an arbitrary number of embedded *while* loops within the body of either S_1 or S_2, or both, it is a simple thing to conclude that the time complexity of our analysis of $S_1; S_2$ (worst case) is the same as that of alternation (computed in Table 3.14), i.e., $O(\ (m.n)^r\ )$.

Next, we consider the time complexity of our GSL flow analysis of protected (or preconditioned) substitutions, **PRE** P **THEN** S **END**, using the following generic algorithm defined in semi-formal pseudocode. Since iterations are not allowed in B machine specifications, we assume the body of the substitution, S, is a simple substitution. (Note: It makes no difference to the complexity analysis if the body is multiple substitution.) This allows us to simply insert the time complexity of simple substitutions, denoted S_{RAM}, which we already estimated to be $O(n)$, into our complexity analysis of protected substitutions.

OVERALL RUNNING TIME: Alternations		
Instructions	**Running Time**	**Comments**
1	$p+1$	$FV(b)$ parsed until $FV(b) = \varnothing$, i.e.,
(i)	p	Executes p times
(ii)	p	Executes p times
(iii)	p	Executes p times
(iV)	p	Executes p times
(v)	1	Executes once
2	$(m.n)^r$	Executes $(m.n)^r$ times
3	1	Executes only once
Total running time = $\mathbf{(m.n)^r + 5p + 3}$		

Table 3.14: Computing overall running time of GSL Analysis of *Alternations*

Protected Substitution (PRE P THEN S END)$^\ell$:

Input: (PRE P THEN S END)$^\ell$:
Output: The total running time cost of protected substitution, denoted *preOut*.
Initialisation: $i := 0,\ p\_Array := \{\},\ output := \{\}$
The RAM Algorithm: (let $m : NAT$ = number of free variables in P)

$parse(P)$
while $(FV(P) \neq \varnothing)$ - - 1
do
begin
$p\_Array[i] := FV(P)_0$ - - (i)
$FV(P)\_out := FV(P)\_out \cup \{(x, p\_Array[i])\}$ - - (ii)
$FV(P) := FV(P) - FV(P)_0$ - - (iii)

```
        i++                                          - - (iv)
    end
return FV(P)_out                                     - - (v)
parse(S)
    S_RAM = n                                        - - 2
return preOut                                        - - 3
```

Assigning 1 cost unit to each instruction in the algorithm, we compute the time complexity of our GSL flow analysis of a protected substitution and tabulate the results in Table 3.15. The estimated total running time is $(5m + n + 3)$. Assuming that $n \geq m$, the time complexity of our information flow analysis of protected substitution is $O(n)$, otherwise it is $O(m)$. Similarly, the time complexity of B GSL substitutions like **ANY**, **SELECT**, etc. is similar to that of protected substitutions considered here, so we do not include the details for the former.

We observe that other than GSL substitutions that could contain arbitrarily large recursive iterations, our information flow analysis is very efficient, as the time cost is generally linear.

Up till this point we have been dealing with information flow between variables defined within standalone machines, refinements and implementations. In Chapter 4, we present our framework for analysing information flow between variables in different machines within a B structured development environment.

OVERALL RUNNING TIME : Protected Substitutions		
Instructions	**Running Time**	**Comments**
1	$m+1$	$FV(P)$ parsed until $FV(P) = \varnothing$, i.e.,
(i)	m	Executes m times
(ii)	m	Executes m times
(iii)	m	Executes m times
(iV)	m	Executes m times
(v)	1	Executes once
2	n	Executes n times
3	1	Executes only once
Total running time = $(\mathbf{5m+n+3})$		

Table 3.15: Computing overall running time of GSL Analysis of *Protected Substitutions*

Chapter 4

Noninterference Flow-Sensitive Structuring Mechanisms in B Machines

4.1 Introduction

In Chapter 3 we presented a rigorous analysis of *intra-machine* flows, i.e., flows between the static[1] parts of the same machine. In this chapter, however, we shall be dealing with *inter-machine* flows, i.e. flows between the static parts of different machines due to modular development via structuring mechanisms and refinement / implementation in the B Method. We begin by discussing existing structuring mechanisms and their visibility rules, and then proceed to show how this could be exploited by an adversary to leak secret information. Finally we conclude the chapter with an analysis and formulation of security conditions that guarantee secure flows between machines within a development.

4.2 Structuring Mechanisms and Visibility of B Machines

It is worth noting that like many other programming / specification languages in Computer Science the Generalised Substitution Language is *pri-*

[1] In this thesis we only consider the machine variables as the static parts of B machines.

marily concerned with *the prevention of unauthorised 'writes'*[2]. By this we mean that a lot of effort is put into ensuring that data within a module is modified in a controlled and assured way. Also, languages are concerned with providing shared access to assets where possible, so long as the integrity and safety of the assets are not jeopardised. Information hiding and access control mechanisms are often used to control how parts of a *modular* software system are modified. However, little has been done from a *language-based* perspective to prevent *unauthorised 'reads'*, i.e., to prevent unauthorised persons from learning secret information within the system via legitimate observations of low security object input and output. For example, a B refinement is not allowed to call *update* operations of the refined machine, whereas, it is allowed to call *inquiry* operations of the refined machine wherefor information could be read. Through such *reads*, there is the possibility of information flow from high security variables in the refined machine to low security variables in the refining machine. Similar insecure flows are possible also with structuring mechanisms. This point will be clearly seen as we discuss the structuring mechanisms in the B Method in this chapter.

We organise our presentation of structuring mechanisms in the B Method into two primary parts, namely: read-only structuring mechanisms (termed RSM), and read-write structuring mechanisms (termed RWSM).

4.2.1 Read-Only Structuring Mechanisms (RSM)

The RSMs in the B Method are the *SEES* and *USES* structuring mechanisms introduced via clauses bearing their respective names. Both RSMs allow shared access to sets, constants, variables and *inquiry* operations of the seen (or used) machine(s). The main difference between both RSMs is that while a *seeing* machine may only read variables either directly or via inquiry operations of the *seen* machine, a *using* machine goes one step further. In addition to ability to read variables in a *used* machine, a *using* machine may impose additional constraints on the variables of the *used* machine, thus these variables are visible in the *INVARIANT* clause of the *using* machine. Other notable differences include:

[2]Some variable *reads* could be prevented by data-hiding / encapsulation mechanisms in some languages, but this does not guarantee secure information flow between variables.

- The *USES* RSM can only be used in abstract machines, whereas the *SEES* RSM can be used in machines, refinements, and implementations, albeit with slight variations in it's visibility in the different frameworks. Table 4.1[3] shows the visibility rules of the *SEES* RSM in both machines and refinements, whereas Table 4.2 shows the visibility rules of the *SEES* RSM in implementations.

- A used machine along with the machine using it *must* be included in one machine to guarantee consistency of data within the development. This is not a requirement for a seen machine, since each seen machine provides it's own proof obligations. A used machine on the other hand does not discharge its own proof obligations.

- As shown in Tables 4.1 and 4.2, *inquiry* operations of the *seen* machine can be called within the operations of a *seeing* machine, refinement, or implementation. To the contrary, the operations of a *used* machine are not visible from within the *using* machine as shown in Table 4.3

- The formal parameters of the *used* machine are fully visible to the *using* machine, but this is not the case for the *seen* machine, refinement or implementation.

4.2.2 Read-Write Structuring Mechanisms (RWSM)

To deal with *read* and *write* dependencies within large developments, the B abstract machine notation provides two RWSMs, namely, the *INCLUDES* and *IMPORTS* clauses. The *IMPORTS* clause is only allowed within an *implementation* framework, whereas the *INCLUDES* clause can only be used in either a *machine* or a *refinement* framework. Thus the visibility rules for the *INCLUDES* clause is the same in both frameworks where they could be employed, as depicted in Table 4.4. Notice, though, that the visibility rules for the *IMPORTS* clause is different, as shown in Table 4.5. The main difference between the visibility of the two clauses is that the variables of *included* machines are visible to the operations of the *including* machine, whereas, the *abstract variables* of *imported* machines are NOT visible to the

[3]Tables 4.1, 4.2, 4.3, 4.4, and 4.5 are adapted from [1].

SEEN MACHINE OBJECTS			SEEING MACHINE / REFINEMENT			
Refinement Visibility	Machine Visibility		INCLUDES	PROPS	INVARIANT	OPERS
		Parameters				
		Sets	×	×	×	×
		Constants	×	×	×	×
		Variables				*read – only*
	Operations					*inquiry*

Table 4.1: Visibility of objects of Seen machine in a machine and refinement

operations of an *importing* implementation.

Note that if the *imported* machines are parameterised, then the *actual* parameters must be provided in the *VALUES* clause of the implementation, and these must satisfy the imported machines' *CONSTRAINTS*. Similarly, *deferred* sets must be fully defined in the *VALUES* clause of the implementation and must satisfy the *PROPERTIES* clause of the refined or imported machine or refinement.

4.2.3 Composite Read-Write Structuring Mechanisms

In this subsection, we describe the *PROMOTES* and *EXTENDS* clauses of B abstract machines. The *PROMOTES* clause can only be used in the presence of either an *INCLUDES* clause (*when used within a machine or refinement*), or in the presence of an *IMPORTS* clause (*when used within an implementation*). In either case, the *PROMOTES* clause lists the *operations* of the *included* (or *imported*) machines that the developer desire to *upgrade* to become full-fledged operations of the *including* machine / refinement (or *importing* implementation). Note, however, that promoted operations remain *native* operations of the *included* (or *imported*) machines, and hence

SEEN MACHINE		SEEING IMPLEMENTATION				
Objects of Seen Machines		**IMPORTS**	**VAL**	**PROPS**	**INIT**	**OPERS**
	Parameters					
	Sets	×	×	×	×	×
	Concrete Constants	×	×	×	×	×
	Abstract Constants			×	×	*loop invariant only*
	Concrete Variables					*read only*
	Abstract Variables					*loop invariant only*
	Operations					*inquiry*

Table 4.2: Visibility of objects of Seen machine within an Implementation

USED MACHINE		USING MACHINE			
Objects of Used Machines		**INCLUDES**	**PROPS**	**INVARIANT**	**OPERS**
	Parameters			×	×
	Sets		×	×	×
	Constants		×	×	×
	Variables			×	*read only*
	Operations				

Table 4.3: Visibility of objects of Used machine within a Using Machine

INCLUDED MACHINE		INCLUDING MACHINE / REFINEMENT			
Objects of Included Machines		**INCLUDES**	**PROPS**	**INVARIANT**	**OPERS**
	Parameters				
	Sets		×	×	×
	Constants		×	×	×
	Variables			×	*read only*
	Operations				×

Table 4.4: Visibility: Included machine objects in a Machine or Refinement

must satisfy their corresponding preconditions. Thus, *PROMOTED* operations do not appear again within the *OPERATIONS* clause of the *including* machine (or *importing* implementation).

In the case where *all* the operations of an *included* or *imported* machine are to be *promoted*, the more concise construct, *EXTENDS* could be used to replace the construct pair *INCLUDES* and *PROMOTES* (or *IMPORTS* and *PROMOTES*). Hence, the *EXTENDS* clause automatically includes (or imports) a machine(s) as well as promotes all the operations of the machine(s) listed within the clause. Thus it (the *EXTENDS* clause) does not require a *PROMOTES* clause within its framework. Hence, the *EXTENDS* clause could be described as a composite RWSM. As was the case with the operations involved in the *INCLUDES*/*PROMOTES* (or *IMPORTS*/*PROMOTES*) pair, all the operations in an *EXTENDed* machine remain *native* operations of the machine(s) wherein they are defined, hence do not appear again within the *OPERATIONS* clause of the *extending* machine, refinement, or implementation.

IMPORTED MACHINE		IMPORTING IMPLEMENTATION				
Objects of Imported Machines		**IMPORTS**	**VAL**	**PROPS**	**INIT**	**OPERS**
	Parameters					
	Sets		×	×	×	×
	Concrete Constants		×	×	×	×
	Abstract Constants			×	×	*loop invariant only*
	Concrete Variables				×	*read only*
	Abstract Variables				×	*loop invariant only*
	Operations					×

Table 4.5: Visibility: objects of an Imported machine in an Implementation

4.3 Security Limitations of existing Structuring Mechanisms

One of the reasons it is particularly difficult to guarantee secure information flow using existing structures and constraints within the B Method is because, as is the case in traditional specification and programming languages, there is no built-in constraint on either standalone machines or machines within a structured development to prevent the flow of secret information to public objects or components of a development. As in many languages, more effort has been concentrated on ensuring that state data is not *modified* by unauthorised persons.

To illustrate this problem, we present in Table 4.6 an example of a machine that *trivially* satisfies Generalised Noninterference, Separability, and a couple of other possibilistic security properties. However, a valid refinement

```
**********************************************************

MACHINE RefPardx
VARIABLES pVar, sVar
INVARIANT pVar ∈ 0..1 ∧ sVar ∈ 0..1
INITIALISATION pVar, sVar := 0, 0
OPERATIONS
    update_pVar ≙
        pVar := 0 [] pVar := 1
    END;
    update_sVar(in_sVar) ≙
        PRE in_sVar ∈ 0..1 THEN
            sVar := in_sVar
        END
    END;
    ss ← get_sVar ≙
            ss := sVar
    END
END

**********************************************************
```

Table 4.6: Simple *secure* abstract machine

of this machine, illustrated in Table 4.7 clearly fails Noninterference due to the fact that the B Method provides no explicit mechanism for controlling the reading of variables based on security classification within a machine or refinement. We assume in examples shown in Tables 4.6 and 4.7 that $pVar$ and $pVarR$ are both intended to be *public* variables, whereas the variables $sVar$ and $sVarR$ are intended to be *secret* variables, hence data is passed to them (the secret variables) only via operation parameters to prevent them from being read from program text. The output variable ss is a secret variable and can only be read in the normal way in B via calls to the operation $ss \leftarrow get\_sVar$. All variables are assumed to be *boolean* variables with value 0 denoting '$false$' and value 1 denoting '$true$'. Notice that the operation that updates the public variable *update_pVar* in the refinement *implicitly* leaks the value of the secret variable $sVarR$ by using it in the test condition of the IF statement. If, however, we run both the machine and the refinement through Atelier B, the B-Toolkit, or other B tools, they will both 'commit' successfully because the refinement is valid by the classical refine-

```
**************************************************************

REFINEMENT RefPardxR
REFINES RefPardx
VARIABLES pVarR, sVarR
INVARIANT pVarR = pVar  ∧  sVarR = sVar
INITIALISATION pVarR, sVarR := 0, 0
OPERATIONS
    update_pVar  ≙
        IF sVarR = 0 THEN pVarR := 0 ELSE pVarR := 1 END
    END;
    update_sVar(in_sVar)  ≙
        PRE in_sVar ∈ 0..1 THEN
            sVarR := in_sVar
        END
    END;
    ss  ←  get_sVar  ≙
            ss := sVar
    END
END

**************************************************************
```

Table 4.7: Insecure *DEMONIC* Refinement of RefPardx

ment relation notwithstanding the insecure flow.

While Table 4.7 illustrates a classic case of the refinement paradox with respect to security properties, discussed in Section 2.4.3, the extension of this example shown in Table 4.8 illustrates information flow leaks due to structuring mechanisms in the B Method. In this example we assume $pVarStr$ is a public variable. It can be seen clearly that due to the composition of the machine $StructPardx$ with $RefPardx$ via the *USES* clause, the variables of $RefPardx$ becomes *visible*, albeit read-only, within the *OPERATIONS* clause of $StructPardx$. Thus, it is a simple matter to copy the secret stored in $sVar$ of the *used* machine $RefPardx$ into the public variable $pVarStr$ of the *using* machine $StructPardx$.

As shown by the visibility rules presented in Section 4.2 above, wherever a structuring clause allows *read* access to variables of an accessible machine,

```
**********************************************************

MACHINE StructPardx
USES RefPardx
VARIABLES pVarStr
INVARIANT pVarStr ∈ 0..1
INITIALISATION pVarStr := 0
OPERATIONS
    ss ← update_pVarStr ≙
        ss ← get_sVar ||
        pVarStr := ss
    END
END

**********************************************************
```

Table 4.8: Information flow due to *USES* clause

either directly or indirectly through inquiry operations of the accessed machine, no further constraint exists between the two machines, except that such accessed variables *cannot* be modified outside the update operations in the native machine. This is a limitation of the B Method whenever there is a need to provide assurance that secret information is not *read* by unauthorised observers. Hence the example presented in Table 4.8 illustrates the need to analyse possibilistic information flow leaks between structured B machines, refinements and implementations to the end of detecting and preventing insecure inter-machine flows. This is the subject of the next section of this thesis.

4.4 Flow Analysis of Structured B Machines

We use the expression '*structure flow analysis*' to refer to our analysis of information flow between B Machines (i.e., inter-machine flows) due to the structuring mechanisms discussed earlier in Section 4.2. Before we present our formal structure flow analysis, though, it is worth noting that we are only concerned here with the possible reading of state variables (either via RSMs or RWSMs). Even where *INCLUDES* or *IMPORTS* clauses are used whereby *update* (write) *operations* of the included (or imported) machines may be called, included (or imported) variables cannot be modified out-

side the local machine within which it is defined. Hence we need not be concerned about secure information '*write-downs*' (i.e., the writing of secret information *down* to a public variable) with respect to the variables of included (or imported) machines. The information flow analysis of the included (or imported) machines at the *local level* is sufficient to detect flows within any included (or imported) machine(s). Since variables of a refined machine are also visible within the *refining* machine, we treat the *REFINES* clause here along with B structuring mechanisms. We summarise below the visibility rules relating to *structure state variables*, i.e., state variables across a structured development in B.

- *SEES:* Variables of seen machines may only be read within the *OPERATIONS* clause of seeing machine(s) or refinement either directly or through *inquiry operations* of seen machines. Within an implementation, though, *concrete* variables of seen machines may be read within the *OPERATIONS* clause of seeing implementation(s) either directly or through *inquiry operations* of seen machines, and in the case of abstract variables, only within the *loop invariant* of seeing implementation(s).

- *USES:* Variables of used machine(s) may only be read within the *OPERATIONS* and *INVARIANT* clauses of the using machine. The *USES* clause is only allowed in machines;

- *INCLUDES:* Variables of included machines may be read within the *OPERATIONS* clause of including machine or refinement either directly or through *inquiry operations* of included machines, and within the *INVARIANT* clause of including machine or refinement;

- *IMPORTS:* Concrete variables of imported machines may be read within the *OPERATIONS* clause of importing implementation either directly or through *inquiry operations* of imported machines, and in the case of abstract variables, within the *loop invariant* of importing implementation. Imported variables may also be read within the *INVARIANT* clause of importing implementation.

- *EXTENDS:* The visibility rules that apply to *INCLUDES* in machines and refinements (and *IMPORTS* in implementations) apply also to

the visibility of variables of an *extended* machine within the *extending* machine, refinement or implementation.

- *REFINES:* Variables of a *refined* machine or refinement are visible within the *INVARIANT* clause of the refining refinement or implementation. They are also visible (read-only) within the operations of a refinement, but not within the operations of an implementation. Only concrete variables are visible within the operations of an implementation.

With the summary of visibility of structure state variables given above we now present our formalisation of the structure flow analysis as a layer over the static intra-machine flow analysis presented in Section 3.4. First, we introduce some extra notation to be used in our structure flow analysis.

Additional Notation:
Given that **A** and **B** denote either machines, refinements, or implementations, we generalise the notion that B SEES or USES or INCLUDES, or IMPORTS or EXTENDS A by saying: 'A is *visible* to B'. Semantically, this means that B is able to read the variables defined in A. Notationally, we write $\mathbf{B} \lessdot \mathbf{A}$ to mean, literally, 'B has *visibility* of A', or 'A is visible to B'. We also make a distinction between the variables occurring in **A** and those occurring in **B**. $\mathbf{Ide_A}$ denotes the set of variables occurring *locally* in A, whereas we use $\mathbf{Ide_B}$ to denote the set of variables *native* to B, and $\mathbf{Ide}_\star$ denotes the composite variable space within the development. Given that $\mathbf{B} \lessdot \mathbf{A}$, information flow due to structuring mechanisms allow $\mathbf{Ide_A}$ to be visible within, say, operation $C_2^{\ell_2}$ defined in **B**. Given that $C_B^{\ell_i}$ ranges over operations of B, where i ranges over natural numbers, we write this notion of visibility as $C_B^{\ell_i} \lessdot \mathbf{Ide_A}$. To ensure secure information flow within the structured machines in the development, we require that the security classification of the variables in $\mathbf{Ide}_\star$ map to a complete lattice, which subsumes the security lattices to which both $\mathbf{Ide_A}$ and $\mathbf{Ide_B}$ are mapped, since ...

$$\mathbf{Ide_A} \cup \mathbf{Ide_B} \subseteq \mathbf{Ide}_\star$$

Given that $\mathbf{B} \lessdot \mathbf{A}$, and writing $\mathbf{Ide}_{\mathbf{B}}^{\lessdot}$ to denote the set of all variables visible within B in the development, we have:

$$\mathbf{Ide}_{\mathbf{B}}^{\lessdot} \supseteq \mathbf{Ide}_{\mathbf{A}} \cup \mathbf{Ide}_{\mathbf{B}} \tag{4.1}$$

Theorem 10 (Transitivity of Structure Dependency Relation (GSL))**.** *Given two machines* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *such that* $\mathbf{B} \lessdot \mathbf{A}$, $\mathbf{Ide}_{\mathbf{A}}$ *denotes the set of variables in A,* $\widehat{X}_A$ *denotes the set of variables assigned to in operation* $C_A^{\ell_i}$ *which ranges over operations of* $\boldsymbol{A}$ *and* $\widehat{X}_B$ *denotes the set of variables assigned to in operation* $C_B^{\ell_j}$*, which ranges over operations of* $\boldsymbol{B}$*. Then*

$$\begin{aligned} &\forall (x_A \in \widehat{X}_A,\ y \in \mathbf{Ide}_{\mathbf{A}}) \ \cdot\ x_A\ \widehat{D}(\ell_i)\ y, \\ &\quad (x_B \in \widehat{X}_B)\ \widehat{D}(\ell_j)\ x_A\ \Rightarrow x_B\ \widehat{D}(\ell_j)\ y \end{aligned}$$

Note: Since the variables defined in machine **A** are visible to the operations in machine **B** (Formula 4.1), we over-approximate (in Theorem 10) over variable dependencies in $C_A^{\ell_i}$ (i.e., $x_A\ \widehat{D}(\ell_i)\ y$) that may *transitively* affect updated variables $x_B \in \widehat{X}_B$ in operation $C_B^{\ell_j}$ of machine **B**. Hence it is conceivable that $\widehat{D}(\ell_j) \supseteq \widehat{D}(\ell_i)$.

PROOF:

Given a set of variables $\{x, y, z\}$ on which a relation '*depends on*', denoted R, is defined, we write $\langle x, y\rangle \in R$ to mean 'x *depends on* y'. By the *transitivity of relations*, $\langle x, y\rangle \in R$ and $\langle y, z\rangle \in R$ implies that $\langle x, z\rangle \in R$. That is, '$x$ *depends on* y and y *depends on* z implies that x *depends on* z. This corresponds to Theorem 10. Theorem 10 informs our formulation of the information flow analysis of structured B machines below. But, first, we introduce more notation.

As in Section 3.4, we write $(\widehat{G}, \widehat{D}) \vDash C^{\ell}$ to denote the notion that the analysis $(\widehat{G}, \widehat{D})$ is adequate for the GSL substitution C^{ℓ}. We write $M_{\star}$ to denote a machine (or refinement or implementation) that has visibility of other machine(s); and $C_{\star}^{\ell_{\star}}$ ranges over operations of $M_{\star}$. Thus $C_{\star}^{\ell_{\star}}$ could be any of the substitutions defined in our GSL semantics (Table 3.2). The label $\ell_{\star}$ refers to the body of $C_{\star}$, and we use this in our framework to identify flows within the *local* body of $C_{\star}$. To collect all possible flows, i.e., local flows as well as flows due to structuring mechanisms, we add an extra label, ℓ, to

$C_\star$. Thus we have $(C_\star^{\ell_\star})^\ell$ as our composite labeling strategy for collecting all possible flows in $C_\star$. We use M_i to range over the machines visible to $M_\star$, with $C_i^{\ell_i}$ ranging over the operations of M_i. We write $\widehat{X}_\star$ to denote the set of variables updated within $M_\star$. If the substitution in a visible machine is a protected substitution, then we write $P_i|C_i^{\ell_i}$ to denote the substitution, where P_i is the *protecting* predicate. We now develop in Table 4.4 below the information flow analysis of structured B machines. Given that $M_\star \lessdot M_i$:

$$
\begin{array}{lll}
(\widehat{G},\widehat{D}) \vDash (C_\star^{\ell_\star})^\ell \text{ in } M_\star \lessdot M_i & & \text{- (unprotected substitutions)} \\
\quad \Leftrightarrow (\widehat{G},\widehat{D}) \vDash C_i^{\ell_i} \wedge (\widehat{G},\widehat{D}) \vDash C_\star^{\ell_\star} \wedge & & \\
\quad\quad \widehat{G}(\ell) \supseteq \widehat{G}(\ell_\star) \cup \widehat{G}(\ell_\star);\widehat{D}(\ell_i) \wedge & & \\
\quad\quad \widehat{D}(\ell) \supseteq \widehat{D}(\ell_i) \cup \widehat{D}(\ell_\star) \cup (\widehat{D}(\ell_\star);\widehat{D}(\ell_i)). & &
\end{array}
$$

$$
\begin{array}{lll}
(\widehat{G},\widehat{D}) \vDash (C_\star^{\ell_\star})^\ell \text{ in } M_\star \lessdot M_i & & \text{- (protected substitutions)} \\
\quad \Leftrightarrow (\widehat{G},\widehat{D}) \vDash P_i|C_i^{\ell_i} \wedge (\widehat{G},\widehat{D}) \vDash C_\star^{\ell_\star} \wedge & & \\
\quad\quad (\bullet \in \widehat{G}(\ell_i) \Rightarrow \widehat{G}(\ell) \supseteq FV(P_i)) \wedge & & \\
\quad\quad \widehat{G}(\ell) \supseteq \widehat{G}(\ell_\star) \cup \widehat{G}(\ell_i) \cup \widehat{G}(\ell_\star);\widehat{D}(\ell_i) \wedge & & \\
\quad\quad \widehat{D}(\ell) \supseteq \widehat{D}(\ell_i) \cup \widehat{D}(\ell_\star) \cup (\widehat{D}(\ell_\star);\widehat{D}(\ell_i)) \wedge & & \\
\quad\quad \widehat{D}(\ell) \supseteq (\widehat{X}_\star \times FV(P_i)). & &
\end{array}
$$

Table 4.4: Information flow analysis of structured B machines

Notice that our structure flow analysis framework for unprotected substitutions in Table 4.4 collects the variables local to $C_\star$ that may affect termination (i.e., $\widehat{G}(\ell_\star)$) as well as flows arising from *visible* dependencies that may affect those variables that affect termination of $C_\star^{\ell_\star}$, i.e., $\widehat{G}(\ell_\star);\widehat{D}(\ell_i)$. In addition, the framework collects the dependencies local to C_i and $C_\star$ as well as transitive structure dependencies, (i.e., $\widehat{D}(\ell_\star);\widehat{D}(\ell_i)$). In the case of protected substitutions, though, we define the scenario where the operation of the *visible* machine is a protected substitution, i.e., $P_i|C_i^{\ell_i}$. Here, the free variables in P_i are collected along with other variables that may affect termination of $(C_\star^{\ell_\star})^\ell$, (i.e., $\widehat{G}(\ell) \supseteq FV(P_i)$). Consequently, there is an implicit flow from $FV(P_i)$ to all updated variables in $C_\star^{\ell_\star}$, i.e., $\widehat{D}(\ell) \supseteq (\widehat{X}_\star \times FV(P_i))$.

NOTE: As is common practice in B, we assume unique variable naming in all machines within the development. Note, too, that the flow analysis for

EXTENDS corresponds to either **INCLUDES** or **IMPORTS**, depending on whether it is employed in a machine, refinement or implementation.

The information flow analysis framework presented above yields the superset of all possible flows of information between the variables in a development where a machine has visibility of another machine. We now present a simple example below to illustrate how our structure flow analysis enhances the traceability of information flow within a machine that has visibility of another machine. Hence if the security classification of a variable is changed, we can easily track other machines that may need to be modified, as flows (due to structuring mechanisms) from the variable may no longer be secure. The analyser thus developed will also enable the developer to review the flow to see if he/she desires the security classification of the variable(s) concerned to '*float*' up in the manner introduced by Hunt and Sands in [75] (see subheading 'Flow-Sensitive Type-Based Approach to Confidentiality' in Section 2.2.2.10).

In the example below, we have a machine, SeenMachine, which updates two machine variables. We also have another machine, SeeingMachine, which has visibility of SeenMachine, and updates a third variable. Notice in SeenMachine that the operation $uu \leftarrow update\_yy(mm)$ simply assigns an arbitrary natural number to the variable yy, and outputs the value on uu, whereas the operation $oo \leftarrow update\_xx(nn)$ adds an arbitrary (non-zero) natural number to yy, assigns the sum to xx and then outputs the value on oo. To analyse the information flow in SeenMachine, we apply the information flow analysis rule for *multiple substitution* presented in Table 3.4b to the body of $oo \leftarrow update\_xx(nn)$. Thus, without loss of generality, $oo \leftarrow update\_xx(nn)$ corresponds to ${}_aC^{\ell}$, $xx := yy + nn$ corresponds to ${}_aC_1^{\ell_1}$ and $oo := xx$ corresponds to ${}_aC_2^{\ell_2}$. Hence, with a slight abuse of notation to enhance clarity, we have

$\widehat{G}(\ell) \supseteq \widehat{G}(xx := yy + nn) \cup \widehat{G}(oo := xx) \;=\; \varnothing$, and
Since we have the precondition $nn \in \mathbf{NAT1}$, then $\bullet \in \widehat{G}(\ell)$
Thus, $\quad \widehat{G}(\ell) \supseteq FV(nn \in \mathbf{NAT1})$, i.e.,
$\widehat{G}(\ell) = \{nn\}$
$\widehat{D}(\ell) \supseteq \widehat{D}(xx := yy + nn) \cup \widehat{D}(oo := xx) \cup (\widehat{X} \times FV(nn \in \mathbf{NAT1}))$
$= \{(xx, yy), (xx, nn), (oo, xx), (oo, yy), (oo, nn)\}$

Next, we apply the same information flow analysis rule for *multiple substitution* in Table 3.4b to the body of $uu \leftarrow update\_yy(mm)$. In this case, $uu \leftarrow update\_yy(mm)$ corresponds to ${}_aC^{\ell}$, $yy := mm$ corresponds to ${}_aC_1^{\ell_1}$, and $uu := yy$ corresponds to ${}_aC_2^{\ell_2}$. Hence, this time, we have

$$\begin{aligned}
&\widehat{G}(\ell) \supseteq \widehat{G}(yy := mm) \cup \widehat{G}(uu := yy) \;=\; \varnothing, \text{ and} \\
&\text{Since we have the precondition } mm \in \mathbf{NAT1}, \text{ then } \bullet \in \widehat{G}(\ell) \\
&\text{Thus,} \quad \widehat{G}(\ell) \supseteq FV(mm \in \mathbf{NAT1}), \text{ i.e.,} \\
&\widehat{G}(\ell) = \{mm\} \\
&\widehat{D}(\ell) \supseteq \widehat{D}(yy := mm) \cup \widehat{D}(uu := yy) \cup (\widehat{X} \times FV(mm \in \mathbf{NAT1})) \\
&\qquad = \{(yy, mm), (uu, yy), (uu, mm)\}
\end{aligned}$$

It follows from the foregoing therefore that the flow analysis of $oo \leftarrow update\_xx(nn)$ yields the set of dependencies:

$$\{(xx, yy), (xx, nn), (oo, xx), (oo, yy), (oo, nn), (xx, mm), (oo, mm)\}$$

We present the definition of SeenMachine below

MACHINE SeenMachine
VARIABLES xx, yy
INVARIANT $xx \in \mathbf{NAT1} \;\wedge\; yy \in \mathbf{NAT} \;\wedge\; xx >= yy$
INITIALISATION $xx, yy \;:=\; 1, 0$
OPERATIONS

$oo \leftarrow update\_xx(nn) =$
PRE $nn \in \mathbf{NAT1}$ **THEN**
$xx := yy + nn \;\parallel\; oo := xx$
END;

$uu \leftarrow update\_yy(mm) =$
PRE $mm \in \mathbf{NAT}$ **THEN**
$yy := mm \;\parallel\; uu := yy$
END
END

We now present SeeingMachine.

MACHINE SeeingMachine
SEES SeenMachine
VARIABLES ww, zz
INVARIANT $ww \in \mathbf{NAT} \ \wedge \ zz \in \mathbf{NAT1}$
INITIALISATION $ww, zz \ := \ 0, 1$
OPERATIONS

$ss \leftarrow update\_ww(pp) =$
PRE $pp \in \mathbf{NAT}$ **THEN**
$ww := pp \ \parallel \ ss := ww$
END;

$rr \leftarrow update\_zz(qq) =$
PRE $qq \in \mathbf{NAT1}$ **THEN**
$zz := xx + ww \ \parallel \ rr := zz$
END

END

We follow with a structure flow analysis of both machines. Notice that $rr \leftarrow update\_zz(qq)$ uses the variable xx defined in SeenMachine on the right hand side of the assignment operator. This is acceptable since SeeingMachine has visibility of SeenMachine, i.e., SeeingMachine $\lessdot$ SeenMachine. Also notice that xx itself depends on other variables defined within SeenMachine. The use of our structure flow analysis enables us to *trace* this dependency trail, as shown below.

Without loss of generality, we only analyse the dependency flow within operation $rr \leftarrow update\_zz(qq)$ of SeeingMachine for illustrative purpose. SeeingMachine corresponds to $M_\star$, SeenMachine corresponds to M_i, $rr \leftarrow update\_zz(qq)$ corresponds to $C_\star^{\ell_\star}$, and $oo \leftarrow update\_xx(qq)$ corresponds to $P_i|C_i^{\ell_i}$ in Table 4.4. Hence,

For global flows, we have:

$$\widehat{G}(\ell) \supseteq \widehat{G}(\ell_\star) \cup \widehat{G}(\ell_i) \cup \widehat{G}(\ell_\star); \widehat{D}(\ell_i) \cup FV(P_i)$$
$$= \{qq\} \cup \{nn\} \cup (\{qq\} \ ; \ \{(xx, yy), (xx, nn), (oo, xx),$$

$$
\begin{aligned}
&(oo, yy), (oo, nn)\}) \cup \{nn\} \\
&= \{qq, nn\}
\end{aligned}
$$

For dependency flows, we have:

$$
\begin{aligned}
\widehat{D}(\ell) &\supseteq \widehat{D}(\ell_\star) \cup \widehat{D}(\ell_i) \cup (\widehat{D}(\ell_\star); \widehat{D}(\ell_i)) \cup (\widehat{X}_\star \times FV(P_i)) \\
&= \{(zz, xx), (zz, ww), (zz, nn), (zz, qq), (rr, zz), (rr, xx), \\
&\quad (rr, ww), (rr, qq), (rr, nn)\} \cup \\
&\quad \{(xx, yy), (xx, nn), (oo, xx), (oo, yy), (oo, nn)\} \cup \\
&\quad \{(zz, xx), (zz, ww), (zz, nn), (zz, qq), (rr, zz), (rr, xx), \\
&\quad (rr, ww), (rr, qq), (rr, nn)\} \ ; \\
&\quad \{(xx, yy), (xx, nn), (oo, xx), (oo, yy), (oo, nn)\} \\
\\
&= \{(zz, xx), (zz, ww), (zz, nn), (zz, qq), (rr, zz), (rr, xx), \\
&\quad (rr, ww), (rr, qq), (rr, nn), (xx, yy), (xx, qq), (oo, xx), \\
&\quad \{(oo, yy), (oo, nn)\} \cup \\
&\quad \{(zz, yy), (zz, nn), (rr, xx), (rr, yy), (rr, nn)\} \\
\\
&= \{(zz, xx), (zz, ww), (zz, nn), (zz, qq), (rr, zz), (rr, xx), \\
&\quad (rr, ww), (rr, qq), (rr, nn), (xx, yy), (xx, qq), (oo, xx), \\
&\quad (oo, yy), (oo, nn), (zz, yy), (zz, nn), (rr, xx), (rr, yy), \\
&\quad (rr, nn)\}
\end{aligned}
$$

4.5 Security Conditions of GSL Structuring Mechanisms

Using the information presented in Section 4.4, we now extend the security conditions C_1 and C_2 given in Definitions 7 and 8 in Section 3.6 to capture secure information flow due to structuring mechanisms in GSL developments as follows:

In the context of a structured development where $\mathbf{B} \prec \mathbf{A}$, given that:

$$\downarrow X \ = \ \{y \in \mathbf{Ide}_{\mathbf{B}}^{\preccurlyeq} \mid (\exists x \in X \subseteq \mathbf{Ide}_{\mathbf{B}})\ classify(y) \leq classify(x)\}$$

and

$$\downarrow X^c \;=\; \mathbf{Ide}_{\mathbf{B}}^{\lhd} - \downarrow X$$

Consequently, our structure security conditions become:

$$\forall z \;\in \downarrow X^c,$$
$$\mathbf{B} \;\lhd\; \mathbf{A} \;\Leftrightarrow\; \mathbf{Ide}_{\mathbf{B}}^{\lhd} \supseteq \mathbf{Ide}_{\mathbf{A}} \wedge \forall x \in \widehat{X}_B \cdot (D_x \cap z = \varnothing) \qquad \text{- - } C_3$$
(extension of C_1)

and given that $\mathcal{L}$ is the security lattice space spanned by the variables in $\mathbf{Ide}_{\mathbf{B}}^{\lhd}$ such that there exists a $\bot \in \mathcal{L}$, then $X_\bot$ is the set of variables such that

$$\forall l \in \mathcal{L} \cdot (classify(X_\bot) = \bot \leq l)$$

We then derive $X_\bot^c = \{y \in \mathbf{Ide}_{\mathbf{B}}^{\lhd} \mid y \notin X_\bot\}$

hence we have:

$$\mathbf{B} \;\lhd\; \mathbf{A} \;\Leftrightarrow\; \forall x \in X_\bot \cdot (X_\bot^c \cap \widehat{G} = \varnothing) \qquad \text{- - } C_4$$
(extension of C_2)

Having presented our extension of the information flow analysis of standalone B machines to inter-machine flows induced by B structuring mechanisms, we present in Chapter 5 a case study, which we analyse with our information flow analyser (a C++ implementation of our information flow analysis of B machines).

Chapter 5

Flow Respecting Developments in B

5.1 Introduction

In *The Origin Of Life, Five Questions Worth Asking* [134], Christopher Sykes was quoted as reporting in [132] that the famous scientist Richard Feynman left this note on a blackboard shortly before his death (in 1988 [146]): "What I cannot create, I do not understand." This thought perfectly articulates the intuition behind the need to "create" an information flow analyser based on our information flow analyses framework, the intent of which is to get a better understanding of how the theoretic framework discussed in Sections 3.4, 4.4 and 4.5 works in practice. In this chapter we present a detailed case-study of a B development and we use our information flow analyses framework to check for insecure information flow between machine variables. We define an information flow policy on the flow relationships between the variables within the development. We then propose a flow-respecting development methodology for stepwise formal developments in B (illustrated in Figure 5.2). The case study shows how abstract B machines may mask the possibility of insecure refinements and how information leakage due to B structuring mechanisms may be prevented in practice using our proposed flow-sensitive development methodology. The case study is developed in the following section.

5.2 IT Reseller Business Monitor: A Case-Study

We make a start here of defining the scope of the case-study used to demonstrate the functionality of the information flow analysis presented in Sections 3.4 and 4.4 of this thesis.

5.2.1 Aims and Objectives

The aim of the case-study presented in this chapter is not to provide a lesson in formal development of software systems using the B Method. Rather, the case study is intended to serve as a *proof of concept* to demonstrate the feasibility and usability of the information flow analysis framework introduced in Sections 3.4 and 4.4 of this thesis. This we intend to accomplish by running the hypothetical B machines, refinements and implementations derived from the case study along with defined information flow policies through the automatic information flow analyser we developed, using C++, to determine whether flows between variables are secure or not.

The objective of the case study is to contrive a software system that includes at least the following B GSL constructs:

- Underspecification, e.g., by means of the **ANY** construct;
- Conditioned substitutions to varying levels of depth, e.g., by means of the **SELECT** and **IF** constructs.
- Protected substitutions to varying levels of depth, e.g., by means of the **PRE** construct
- Refinements of defined B machines
- Implementation of defined B machines or refinements.
- Structuring mechanisms, e.g., by means of the **SEES** clause.

We then pass the contrived system to our information flow analyser to demonstrate how information flow between variables can be *automatically* tracked at every step of the B development cycle and the developer alerted to insecure flows in the different scenario investigated. We now provide a brief background information to the case study in Subsection 5.2.2.

5.2.2 Background

I was born in Okitipupa, a small town in the south-western region of Nigeria, which transliterated into English, means '*mound red*'. Hence we employ the hypothetical name Redmound-IBM to refer to the IT reseller Business Monitor. An IT reseller is a retail enterprise that deals in computers and associated equipments, components, off-the-shelf applications, and PC maintenance and upgrades. Some well known IT Resellers in the UK are Mesh Computers, PC World, MacWorld, Gultronics, and Micro Anvika. An IT hardware and/or software vendor is an enterprise that produces hardware and/or software tools for sale generally through a distribution channel. Well-known hardware vendors include Toshiba, Sony, HP, Samsung, Apple etc., and well-known software vendors include Microsoft, Adobe, Sage, Apple etc. IT resellers obtain their wares mostly from distributors and sometimes directly from vendors. The vendors also organise training events and promotions for IT Resellers' staff.

The core business of the IT Reseller could be broadly divided into three departments, namely: Administration, Accounts, and Sales/Technical. The Administration department handles general administrative, personnel and training duties. The Accounts department deals with salaries, payments, purchases, client accounts and taxation and corporate reports. The Sales/ Technical department deals with shop floor sales, IT support, computer upgrades and builds, online/mail order sales and distribution, network and application support, vendors and distributors liaison, stock management, adverts and promotions.

Most IT Resellers store some of their business data in bespoke database systems, commonly developed using Microsoft Access or Microsoft SQL. Some other data are retrieved from web forms and processed sometimes manually and sometimes by the database. Other data, e.g., telephone order details, are paper-based. We present here an informal list of the business processes by department within the IT Reseller enterprise, and stipulate whether they are database-driven (dBD) and/or manual (Mnl):

Administration:
Personnel_Records (dBD): stores and manipulates the personal records of

employees, e.g., name, position, start date, home address, holidays, etc.
Inventories and Overheads (Mnl): records and monitors company assets, e.g., servers, pcs, printers, stationeries, etc.
Training_and_management (Mnl): identifies training needs and organises staff training, deals with commendation and promotion, attendance, and general management tasks.

Accounts:
Salaries and loans (dBD): tracks and monitors attendance, salaries and loans to employees, NI, PAYE and other deductions.
General Accounting (dBD and Mnl): manages corporate accounts and reports, e.g., VAT, Profit and Loss accounts, etc.; tracks purchases for stock replenishment and internal company use, reviews and meets staff payment orders.
Client_Accounts (dBD and Mnl): registers and monitors client credit and debit accounts, processes credit/debit card orders, etc.

Sales/Technical:
Sales (dBD and Mnl): deals with all shop floor, mail order and online sales. Organises and delivers promotional sales events in association with product vendors and distributors.
Stock Management (dBD and Mnl): contacts vendors and/or distributors for orders when stock is low. Recommends alternative products to customers when requested product is discontinued.
Technical (Mnl): provides before- and after-sales service to customers; maintains local network and provides application support.

With the hierarchical free-flow diagram in Figure 5.1, we illustrate a summary of the conceptual top level system behaviour described in this section, which sets out the scope of the case study.

To keep track of the business data within the IT Reseller enterprise, we aim to interact with the different data storage and processing systems within the enterprise, and monitor business rules and performance standards that need to be met. Redmound-IBM is the software tool we aim to develop, using the B Method, to meet this need. We will then employ our Information Flow

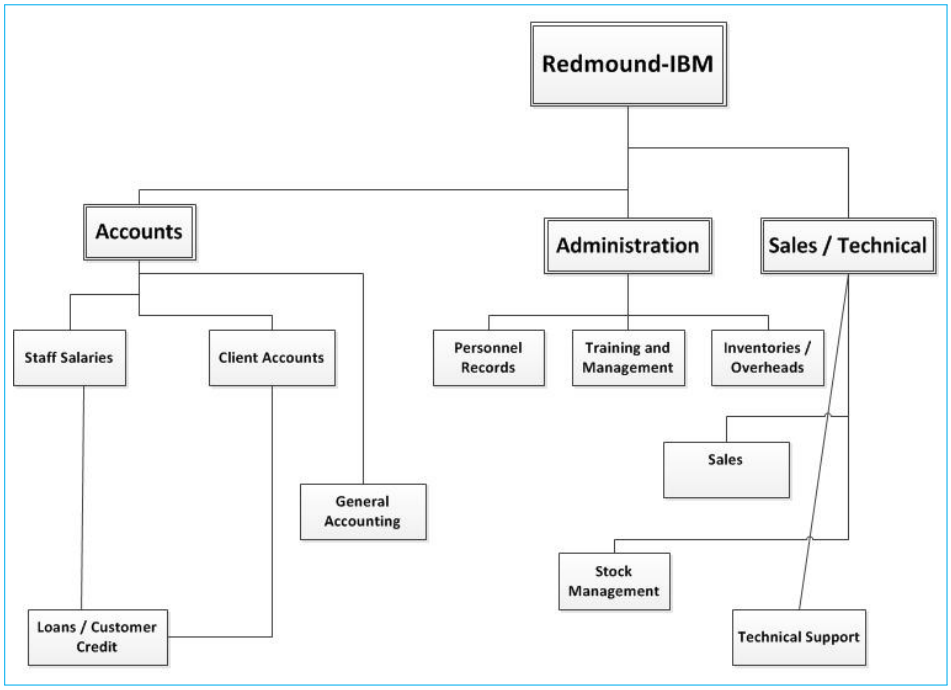


Figure 5.1: Redmound-IBM: Top Level Departmental Hierarchy

Analyser, developed using C++, to *automatically* check for the *possibility* of insecure flow of information between different variables within the development.

We propose a methodology for information flow sensitive developments using the B Method. This methodology involves iteratively running B machines and defined information flow policies through our flow analyser and modifying the machines[1] and/or information flow policy files as necessary until machines with the desired secure information flow are realised. Thereafter the developer can proceed to the next level in the development cycle with the *assurance* that all possible information flow between variables at the current level are secure. The schematics in Figure 5.2 depicts our proposed approach to information flow sensitive development in B. (NOTE: In principle, the methodology illustrated in Figure 5.2 can be applied to other formal methods of software development.)

[1] 'machines' in this context can also be *refinements* or *implementations*.

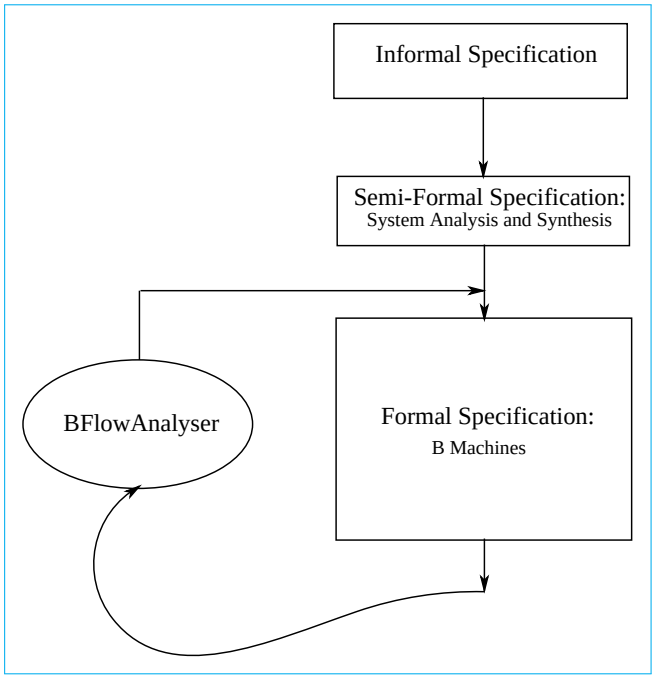


Figure 5.2: An Approach to Flow-Sensitive Development in B

Having introduced the scope of the case-study, we present an Informal Specification of Redmound-IBM in the following subsection.

5.2.3 Redmound-IBM: An Informal Specification

In this section we present a *top level description of system behaviour*.
Redmound-IBM monitors the data generated from business activities and reacts to ensure safe and secure standards are met. Business rules are defined to control critical business operations, and these rules are triggered by data input to the application. Operations monitored by Redmound-IBM include product stock levels, customer credit accounts, transactions and product release, staff salaries and loans, administrative overheads, employee performance and training, corporate reports and taxation. We describe each of these core operations below.

Product Stock Levels
Redmound-IBM receives information about current stock level of each prod-

uct at the end of each business day and applies one of the following business rules:

- For products costing less than £50, if stock level is less than 8, order 8 more items;
- For products costing between £50 and £100, if stock is less than 5, order 5 more items;
- For products costing more than £100, if stock is less than 3, order 3 more items.

Customer Credit Accounts
The IT Reseller makes a 30-day interest free credit facility available to premium customers, subject to the following business rules to be monitored by Redmound-IBM:

- Premium customers are allowed a 30-day interest free facility of not more than £100 on purchases;
- Premium customers must fully pay up before the expiry of the 30-day interest-free period. Otherwise, interest will be charged daily at a rate of 30% APR; further interest-free facilities will be suspended and customer's name will be added to the list of customers with credit default.

Transactions and Product Release
For cash and debit/credit card transactions on the shop floor, goods purchased may be released to the customers on successful completion of transaction. If customer desires product to be shipped to any UK address provided, this will be subject to a shipping surcharge based on bulk and/or weight of item. Online orders and mail orders have to meet additional requirements before product can be released. The business rules guiding product release for all transactions are itemised below:

- Where shipping to any UK address is required, a shipping surcharge of £5 must be paid, if product weighs less than 2kg; £8 shipping surcharge if product weighs 2kg or more but less than 5kg, and £10 shipping surcharge applies to products weighing 5kg or more, but not more than 10kg; special shipping arrangements will be made for products weighing more than 10kg;

- For shop-floor cash and debit/credit card transactions where client requests shipping to any UK address, the product may be released only on payment of appropriate shipping surcharge as described above;
- For online and mail order transactions, allow up to 5 working days for completion of debit and/or credit card transactions;
- For online and mail order transactions, product may only be released on verification of payment for goods and appropriate shipping costs as described above.

Staff Salaries and Benefits
Staff salaries are paid four-weekly in arrears of work done. A four-weekly payment period is termed a payment cycle. One week is constituted of five work days and two rest days. Staff benefits include interest-free loans and discounts on products purchased by staff members for personal use. Employees may be given a loan facility up to 25% of four weekly salary. Repayments are spread over a period of 6 salary payment cycles. Redmound-IBM monitors staff salaries, loans and discounts, and enforces the following business rules:

- Where an employee requests a loan facility, the total of outstanding loan, if any, and amount currently requested must not exceed 25% of four-weekly salary. Otherwise reject request;
- Repayments of any outstanding loan to employees must be made with every payment cycle;
- A discount of 5% is applied to all products purchased by staff members for personal use.

Administrative Overheads
The IT Reseller operates a petty cash imprest system, whereby £100 is placed in a petty cash float at the beginning of each month towards stationeries and other sundry business expenses. At the end of the month, any amount spent from the float is replenished. Redmound-IBM checks the amount in petty cash float at the end of every month and replenishes float according to the following business rules:

- Set petty cash float at the beginning of every month to £100;

- Check petty cash float at the end of the month, and if amount in float is less than £100, add more cash to make it up to £100.

Employee Performance and Training
A thirteen payment cycle period is termed a payment year. Employees are allowed four weeks paid annual leave, hence one payment year contains 240 (i.e., (13×4×5) - 20) work days. Employee performance measurement metrics and training needs, tracked by Redmound-IBM, are defined in the business rules below:

- An employee is adjudged "outstanding" if an absence record of not more than 3 days is achieved in the payment year preceding the end of the business year. Salary will be increased by 3% for the *next payment year* (npy);
- An employee is adjudged "okay" if an absence record of not more than 4 or 5 days is achieved in the payment year preceding the end of the business year. Salary will be increased by 2% for the npy;
- An employee is adjudged "Underachieved" if an absence record of 6 or 7 days is achieved in the payment year preceding the end of the business year. For absence record of 6 or 7 days in the payment year, salary will be increased by 1% for the npy and *warning* given;
- For any absence record more than 7 days in the payment year, there will be no salary increase for the npy, and disciplinary action must be taken;
- Employees are required to attend one training/refresher session in one payment year.

Corporate Reports and Taxation
Corporate reports such as profit and loss accounts annual tax returns, value added tax (VAT) returns and claims, etc., are checked to be sure they are correctly completed and submitted by the due date.

- A VAT of 20% shall be added to all sales and service charges. The cumulative amount received as VAT shall be remitted to the Inland Revenue at the end of every payment cycle.

- Ensure all tax and National Insurance contributions deducted from employee salaries are remitted to the Inland Revenue at the end of each payment cycle;
- Ensure annual tax returns are completed before the October 31 deadline.

In Section 5.2.4, we analyse further the intra- and inter-departmental flows as well as the Business-to-Business (B2B) interactions between Redmound-IBM and its customers, vendors and distributors by means of a business process model. We thereby develop a more granular representation of the system and relationships (visibility) between the different participants[2] in the business process.

5.2.4 Redmound-IBM: Business Process Model Analysis

In this section, we add more detail to the top level description of the system introduced in Section 5.2.3. To this end, we employ a diagrammatic business process modelling and specification approach using a current industry-standard Business Process Modelling Notation (BPMN 2.0)[3]. This diagrammatic approach is simple, yet sufficiently precise to make understanding of the internal business processes (within Redmound-IBM) and interaction with external participants clear to business users and software developers alike. Broadly speaking, the BPMN notation comprises the following elements:

- Activities (or Tasks): a process activity is a specified *unbroken* piece of work performed within a process. We present the different types of BPMN 2.0 tasks in Figure 5.3;
- Events: an event is a significant occurrence, i.e., something that happens, within a process. Events are generally classified into three types, namely: *Start Events, Intermediate Events*, and *End Events*, which by their respective names are self-explanatory. We present further subdivisions of events in the tables in Figures 5.3 and 5.4;

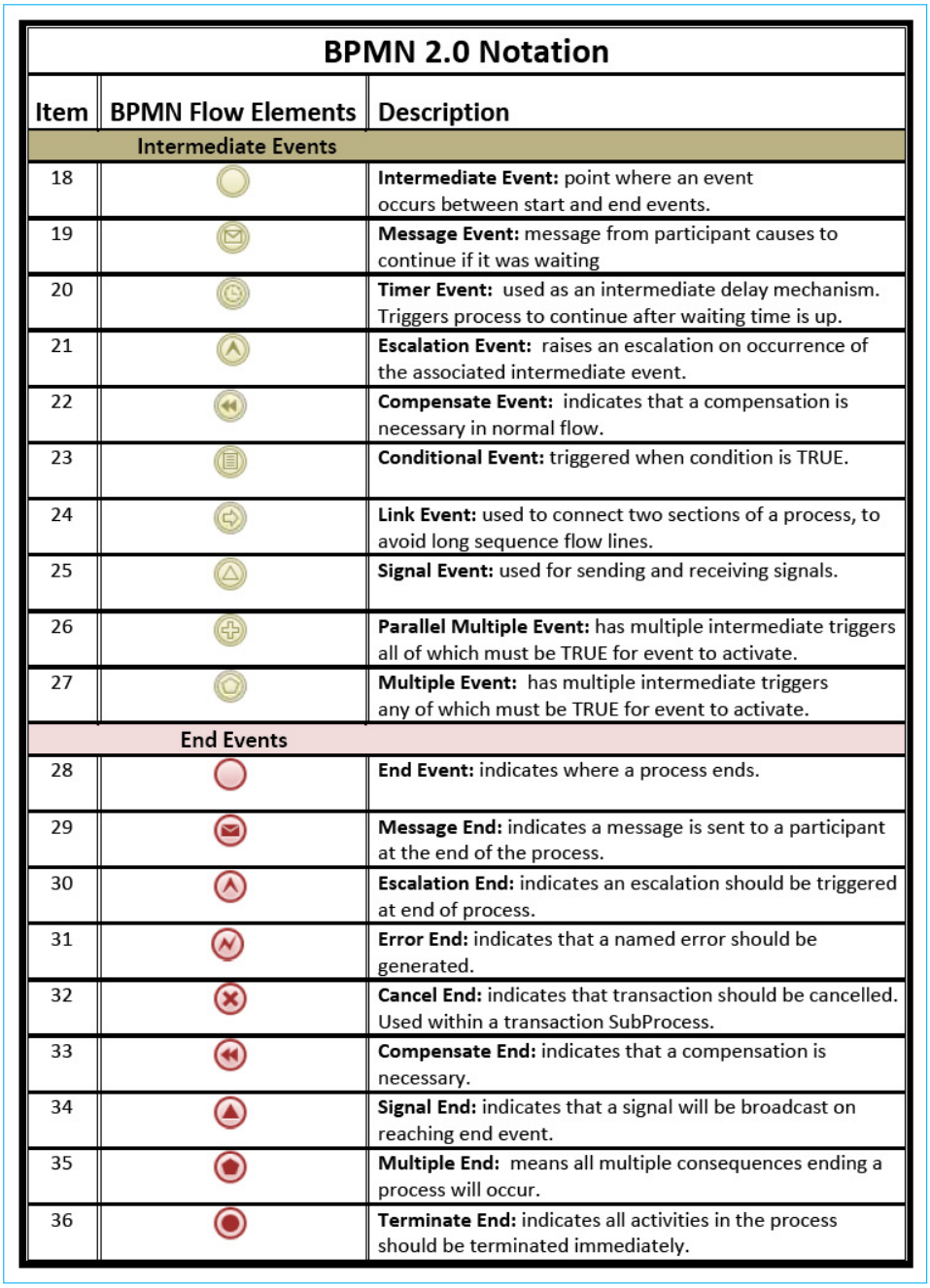
[2] "Participants" in the context of this thesis refers to those roles, departments or organisations that play a part in the performance of the overall process.

[3] BPMN is used to create a standardised bridge for the gap between conceptual business process design and process implementation [117].

BPMN 2.0 Notation

Item	BPMN Flow Elements	Description
	Tasks	
1		**Task** with identifier name embedded. A task is an atomic activity within a process.
2		**User Task:** human performer performs task with the aid of a software tool
3		**Service Task**: performs some service, e.g. web service or some automated application.
4		**Receive Task**: waits for a message from an external participant. Completes when message is received.
5		**Send Task:** sends a message to an external participant. Completes when message is sent.
6		**Script Task:** executed by a business process engine based on script written by modeller / implementer.
7		**Manual Task:** performed without using a business process execution engine or any application.
8		**Business Rule Task:** a means for the process to give input to a business rules engine and get some output.
9		**SubProcess:** an activity that contains other activities. SubProcess sees parent's global data.
10		**Reusable SubProcess:** wrapper for the invocation of a global process within the execution.
	Start Events	
11		**Start Event**: point of entry to process flow. In sequence Flow, no incoming sequence can connect to start event.
12		**Message Start:** message from participant triggers start of process.
13		**Timer Start:** triggers start of process at some specified time, date or cycle.
14		**Conditional Start:** triggers process when a condition is TRUE. Condition must be FALSE for process to stop.
15		**Signal Start**: triggers process on receipt of a broadcast signal from another process.
16		**Parallel Multiple Start**: multiple triggers required Before process can be triggered.
17		**Multiple Start:** any one of multiple ways to trigger a process required to start the process.

Figure 5.3: BPMN 2.0 Notation: Tasks and Start Events.



BPMN 2.0 Notation		
Item	**BPMN Flow Elements**	**Description**
	Intermediate Events	
18		**Intermediate Event:** point where an event occurs between start and end events.
19		**Message Event:** message from participant causes to continue if it was waiting
20		**Timer Event:** used as an intermediate delay mechanism. Triggers process to continue after waiting time is up.
21		**Escalation Event:** raises an escalation on occurrence of the associated intermediate event.
22		**Compensate Event:** indicates that a compensation is necessary in normal flow.
23		**Conditional Event:** triggered when condition is TRUE.
24		**Link Event:** used to connect two sections of a process, to avoid long sequence flow lines.
25		**Signal Event:** used for sending and receiving signals.
26		**Parallel Multiple Event:** has multiple intermediate triggers all of which must be TRUE for event to activate.
27		**Multiple Event:** has multiple intermediate triggers any of which must be TRUE for event to activate.
	End Events	
28		**End Event:** indicates where a process ends.
29		**Message End:** indicates a message is sent to a participant at the end of the process.
30		**Escalation End:** indicates an escalation should be triggered at end of process.
31		**Error End:** indicates that a named error should be generated.
32		**Cancel End:** indicates that transaction should be cancelled. Used within a transaction SubProcess.
33		**Compensate End:** indicates that a compensation is necessary.
34		**Signal End:** indicates that a signal will be broadcast on reaching end event.
35		**Multiple End:** means all multiple consequences ending a process will occur.
36		**Terminate End:** indicates all activities in the process should be terminated immediately.

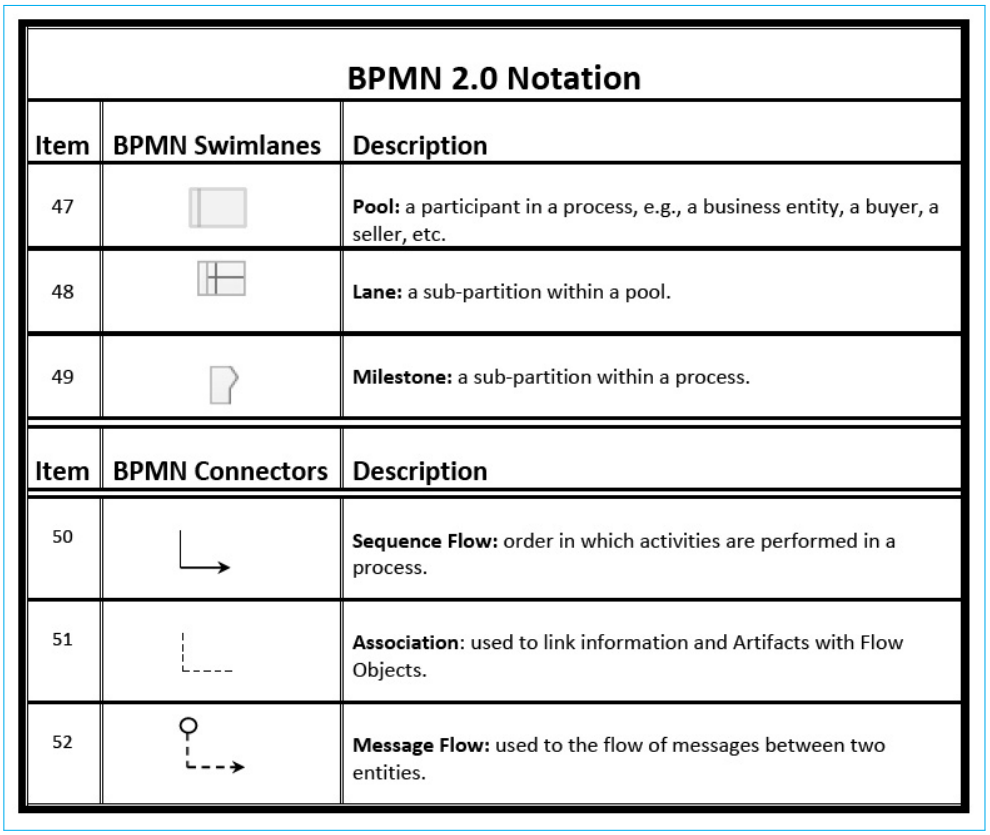
Figure 5.4: BPMN 2.0 Notation: Intermediate and End Events.

BPMN 2.0 Notation

Item	BPMN Flow Elements	Description
	Gateways	
37		**Exclusive Gateway:** decision points within a Business process where alternative paths are open.
38		**Event-Based Gateway:** branching point in a process where choice is based on events at that point.
39		**Exclusive Event-Based Gateway:** each occurrence of a subsequent event starts a new process instance.
40		**Parallel Event-Based Gateway:** other events in the gateway not disabled when one event is triggered.
41		**Inclusive Gateway:** based on conditional expressions. One condition does not exclude evaluation of others.
42		**Complex Gateway:** used to handle situations not easily handled by other gateway types.
43		**Parallel Gateway:** mechanism used to create and synchronise parallel flow.
Item	**BPMN Artifacts**	**Description**
44		**Group:** used to group elements of a diagram informally.
45		**Annotation:** additional text for the reader.
46		**Formatted Text:** allows insertion of rich text into diagram.

Figure 5.5: BPMN 2.0 Notation: Gateways and Artifacts.

- Decisions (or Gateways): Gateways are like forks in a road. They are locations within a business process where decisions have to be made between alternative (multiple) sequence flow paths. The types of Gateways defined in the BPMN 2.0 Specification are summarised in the table in Figure 5.5;
- Artifacts: an artifact is a non-flow object designed to add useful information (e.g., textual or graphical information) to BPMN diagrams. Figure 5.5 shows some commonly used artifacts in BPMN;
- Swimlanes: used to visually distinguish the various sub-processes within a business a process, e.g, by department or function. Swimlanes (par-



BPMN 2.0 Notation

Item	BPMN Swimlanes	Description
47		**Pool:** a participant in a process, e.g., a business entity, a buyer, a seller, etc.
48		**Lane:** a sub-partition within a pool.
49		**Milestone:** a sub-partition within a process.
Item	**BPMN Connectors**	**Description**
50		**Sequence Flow:** order in which activities are performed in a process.
51		**Association**: used to link information and Artifacts with Flow Objects.
52		**Message Flow:** used to the flow of messages between two entities.

Figure 5.6: BPMN 2.0 Notation: Swimlanes and Connectors.

ticularly, *pools*) are also used to distinguish B2B participants in a process. Further details of swimlanes are given in Figure 5.6;

- Flow Connectors: as the name implies, are used to link other BPMN elements together in a meaningful way. The types of flow connectors defined in BPMN 2.0 are illustrated in Figure 5.6.

Using the BPMN 2.0 notations summarised in Figures 5.3, 5.4, 5.5, and 5.6, we are now ready model the architecture of Redmound-IBM, from the informal specification given in Section 5.2.3. We will begin with a top-level collaboration model showing *Sequence Flows* as well as *Message Flows*. We will then refine the top-level diagram further to show details of the embedded sub-processes. Finally we will produce a *System Catalogue*, which we will use as the basis of our work in Section 5.2.5. Notice in the collaboration

diagrams to follow that we adopt the convention of representing *sequential flows* between process elements with horizontal connectors, whereas *message flows* between participants are represented with vertical connectors between *pools*.

5.2.4.1 Top-Level Business Process Collaboration Analysis.

Recall that our information flow analysis framework is concerned with the flow of information between machine variables in a B development. Consequently, in modelling the case study to be analysed by our information flow analyser, we are keen to track both sequence and message flows between the elements and participants respectively of Redmound-IBM.

Thus, we model a generic Customer as a participant in Redmound-IBM business process, i.e., we provide a dedicated pool to the activities, events and decisions typical of a customer's interaction with the IT Reseller business monitor. Similarly, we model distributors, vendors and other suppliers collectively as a participant with one pool representing this group. To facilitate the analysis of inter-departmental message flows, we also model each of the three core departments in Redmound-IBM as participants in the system. Hence we have five pools, and are able to develop a collaboration diagram showing the (*vertical*) message flows between the various participants as shown in Figure 5.7. For example, the *Sales/Technical* process is triggered by the *Order Product* task in the *Customer* process, while the *Distributors/Vendors/Suppliers* process is triggered by the *Manage Stock* sub-process in the *Sales/Technical* process. On the other hand, the *Transaction and Product Release* sub-process within the *Sales/Technical* process provides feedback to the *Make Payment* and *Contact Reseller* tasks within the *Customer* process.

Notice that we leave the *General Accounting* function within the *Accounts* department out to streamline the model and avoid the need for an additional participant (pool) for interaction with the Inland Revenue. This does not in any conceivable way affect the case study. Communication between the elements within each pool is by means of (*horizontal*) sequence flows. For example, the *Administration* process, on being triggered when a new employee starts and at the end of the business year, passes sequence flow

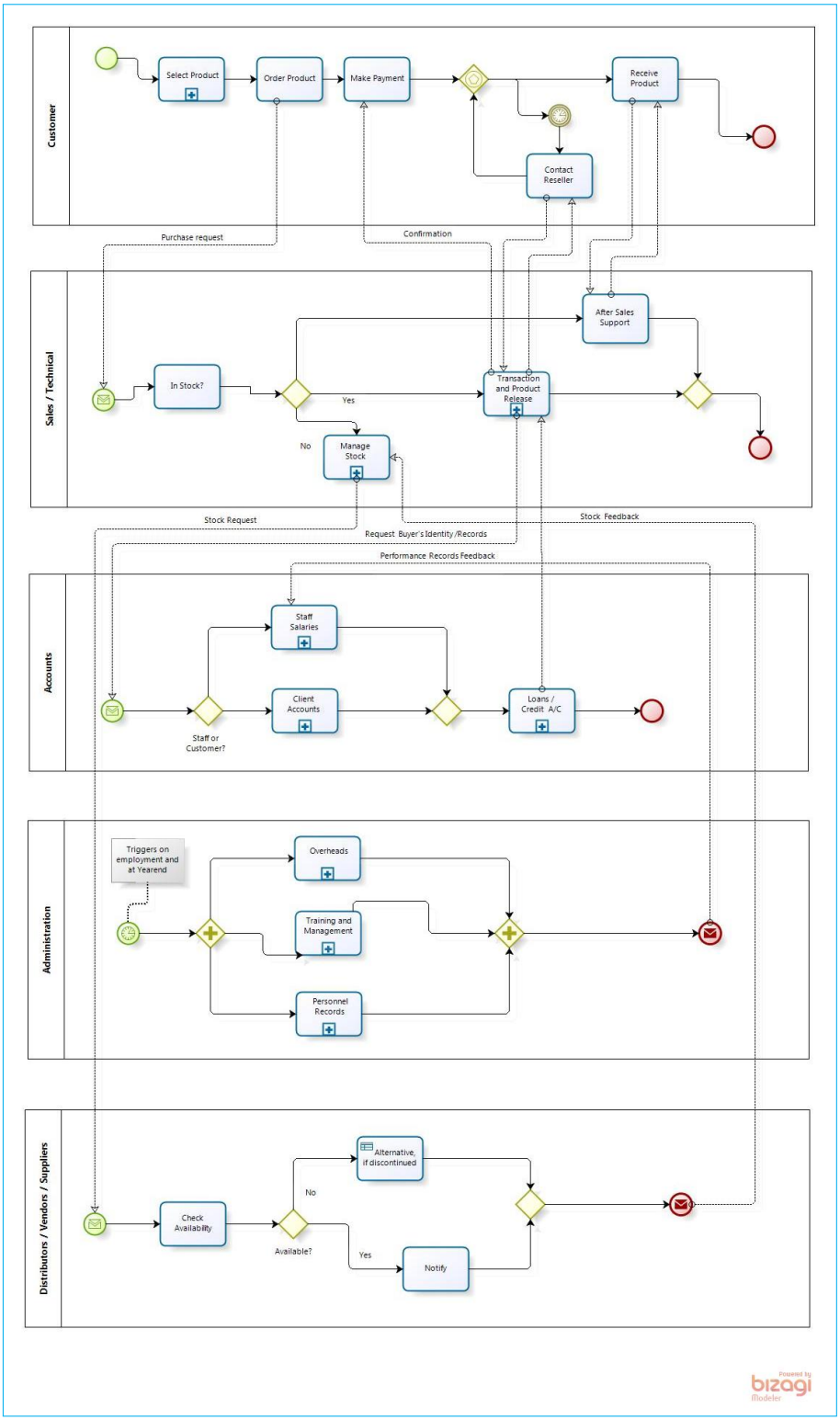


Figure 5.7: Redmound-IBM: Process Collaboration Diagram.

to either the *Overheads*, *Training and Management*, or *Personnel Records* sub-processes.

5.2.4.2 Second-Level Business Process Analysis.

In this section, we provide a more fine-grained analysis of the sub-processes introduced in the collaboration diagram in Figure 5.7. These sub-processes are listed below, viz:

- Select Product
- Manage Stock
- Transaction and Product Release
- Staff Salaries
- Client Accounts
- Loans/Credit Accounts
- Overheads
- Training and Management
- Personnel Records

To begin with, we illustrate the *Select Product* sub-process in Figure 5.8. The sub-process starts with the customer searching the product database via a client interface for availability of product. He then makes a selection, if product sought is available, otherwise the system recommends an alternative, which the customer could accept or reject. Hence there are two possible termination events to the sub-process.

We now present the next sub-process, *Manage Stock* in Figure 5.9. The *Manage Stock* sub-process checks the vendor database for discontinued products, and updates the local database accordingly. The *Monitor Local Stock Database* business rule task then updates product stock according to the decision logic described in Table 5.1.

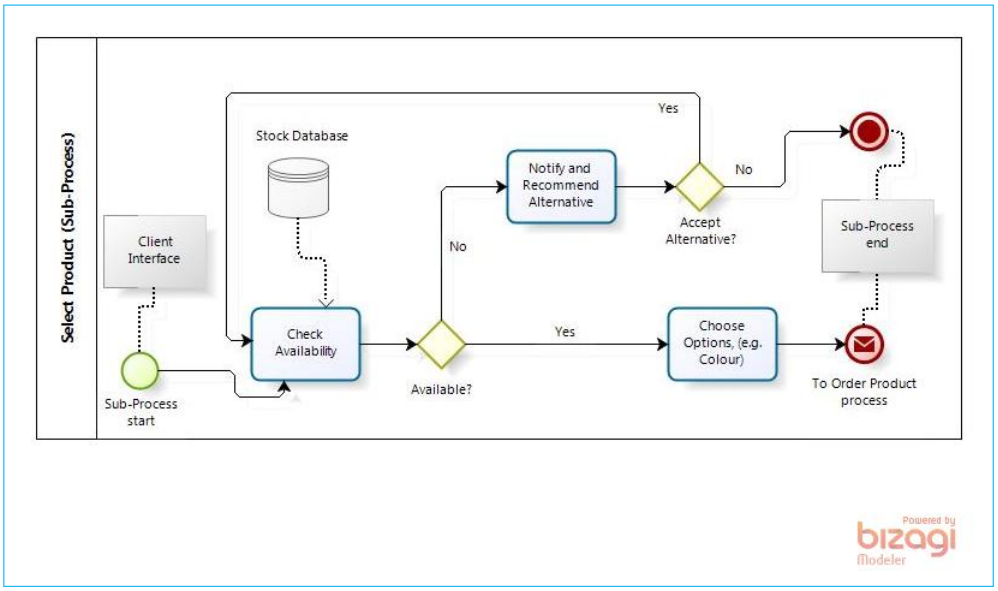


Figure 5.8: Redmound-IBM: Select Product Sub-Process Diagram.

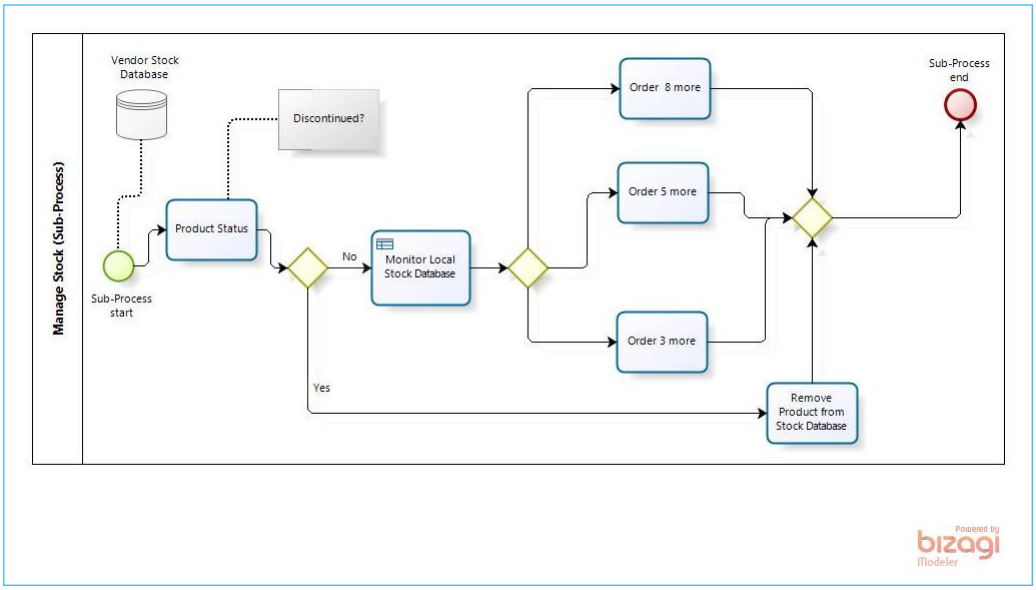


Figure 5.9: Redmound-IBM: Manage Stock Sub-Process Diagram.

Monitor Local Stock Database: Decision Table			
Cost < £50 and **Stock < 8**	**£50 ≤ Cost ≤ £100** and **Stock < 5**	**Cost > £100** and **Stock < 3**	**Decision**
Yes	No	No	Order 8 more
No	Yes	No	Order 5 more
No	No	Yes	Order 3 more

Table 5.1: Monitor Local Stock Database: Decision Table

Figure 5.10 illustrates the *Transaction and Product Release* sub-process. Whether the customer decides to collect product from store or have it delivered, the product needs to be prepared, hence we use an *Inclusive Gateway* to ensure the *Pick and Pack* task is always executed. The *Ship Product* business rule task is used to determine shipping cost based on the decision logic in Table 5.2.

Ship Product: Decision Table				
Weight < 2kg	**2kg ≤ Weight < 5kg**	**5kg ≤ Weight ≤ 10kg**	**Weight > 10kg**	**Decision** (Postage)
Yes	No	No	No	£5
No	Yes	No	No	£8
No	No	Yes	No	£10
No	No	No	Yes	Arrange cost

Table 5.2: Ship Product: Decision Table

The next sub-process we expand on here is the *Staff Salaries* sub-process of the *Accounts* participant. We present this in Figure 5.11. The business rule task *Update Loan* checks that (outstanding Loan + Loan Request) ≤ 25% of Salary, and based on the outcome, the requested loan is either granted or refused. Hence we use an *Exclusive Event-Based Gateway* following the business rule task. If the loan is granted, then the new repayment amount must be recalculated based on the new outstanding loan. Any outstand-

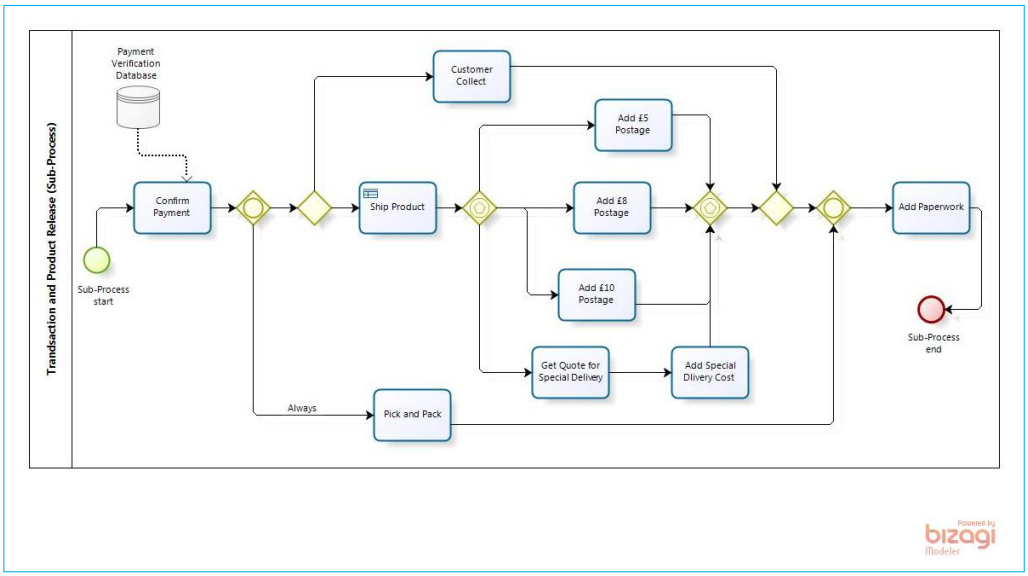


Figure 5.10: Transaction and Product Release Sub-Process Diagram.

ing loan repayment is then deducted before the final salary is paid out to the employee concerned. If, however, an employee has no outstanding loan, then the process ensures no deductions are taken from his/her salary. (Note: We pointed out in Section 5.2.4.1 that we do not model the interaction between Redmound-IBM and the Inland Revenue. Hence, we assume the *Staff Salaries* sub-process is run only after all taxes (PAYE) and National Insurance contributions have been deducted.)

Table 5.3 describes the decision logic used to model the staff loans functionality of Redmound-IBM. The decision logic is based on the informal specification described earlier in Section 5.2.3.

We next expand the *Performance Related Pay* sub-process of the *Staff Salaries* sub-process as illustrated in Figure 5.12. Notice that we use a Timer Start for the sub-process, reason being that the performance related pay is calculated once a year; hence the associated activities are time-triggered at the end of the business year.

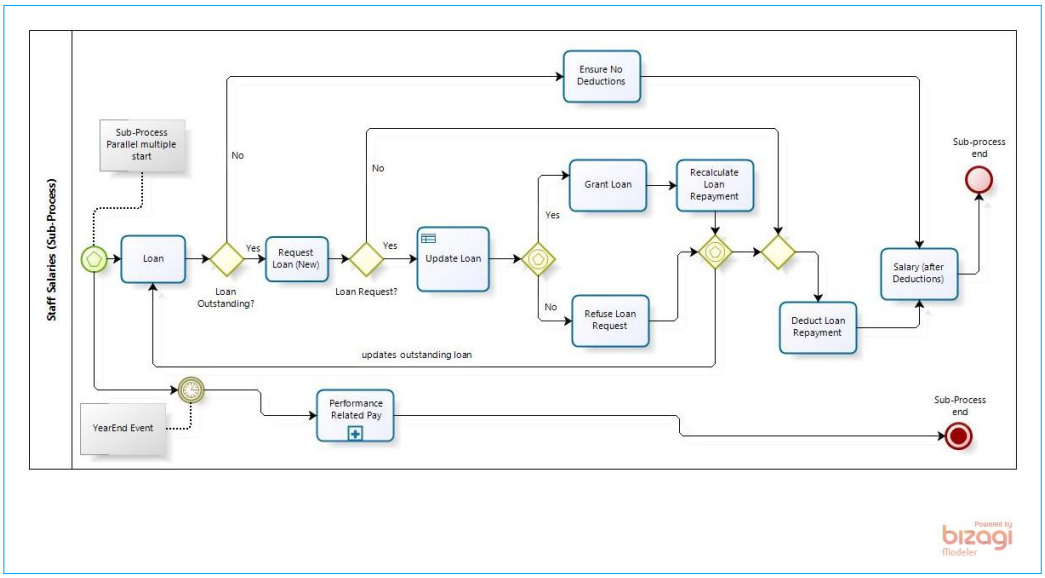


Figure 5.11: Redmound-IBM: Staff Salaries Sub-Process Diagram.

The business rule task *Calculate PRP* is determined by the decision logic presented in Table 5.4.

The next sub-process of the collaboration diagram in Figure 5.2.4.1 that we expand here is the *Client Accounts* sub-process, and this we illustrate in Figure 5.13. Notice that only Premium Customers are allowed to request credit in line with our informal specification in Section 5.2.3.

The business rule *Request Credit* monitors the criteria for granting / refusing credit request to customers based on the policy described in Section 5.2.3. We summarise the decision logic behind *Request Credit* in Table 5.5.

We now expand the sub-process *Loans/Credit Accounts* as shown in Figure 5.14. The feedback to employee and customer is dependent on the decision logic within the business task rules *Check loan from Staff Salaries Sub-Process* and *Check Customer Credit from Client Accounts Sub-Process* respectively. We combine the decisions based on both business task rules in Table 5.6.

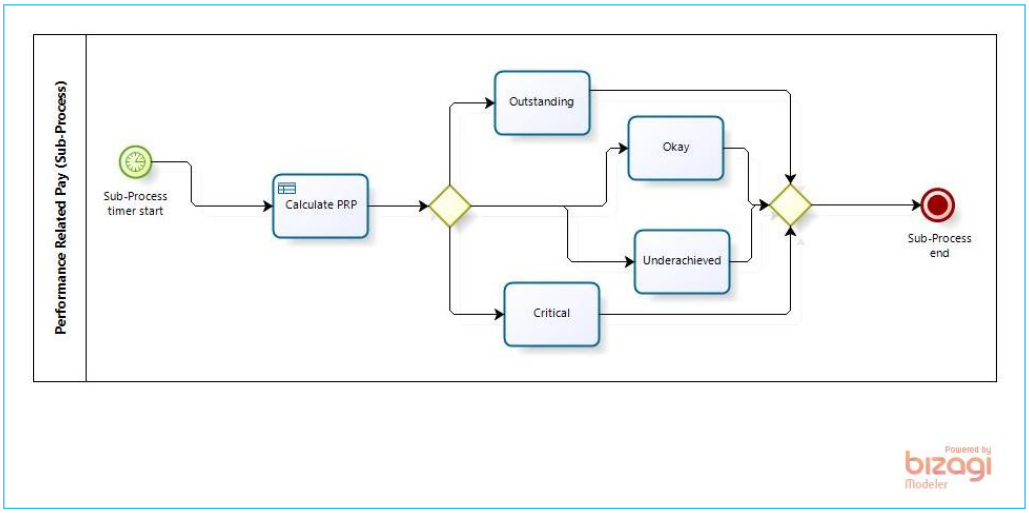


Figure 5.12: Performance Related Pay Sub-Process Diagram.

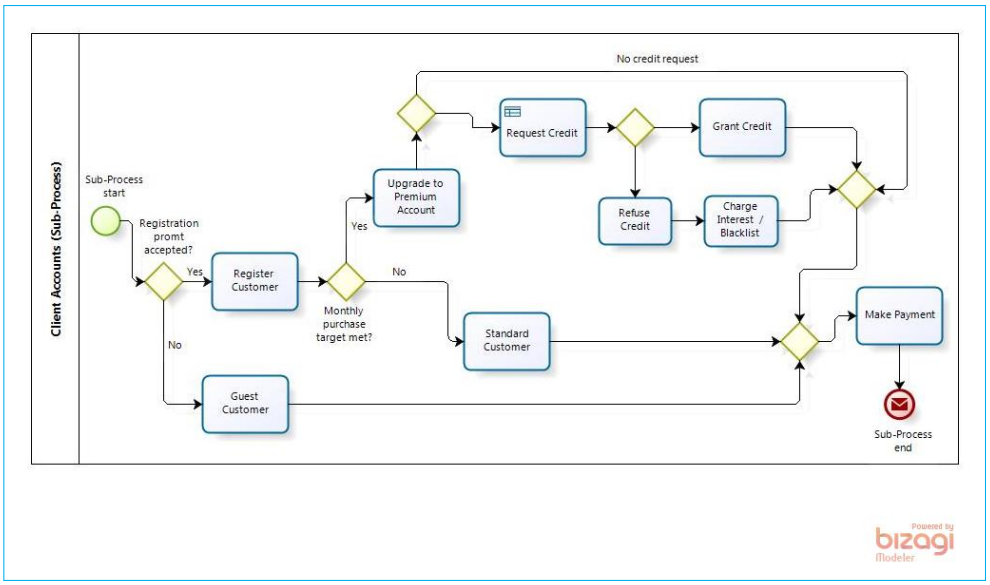


Figure 5.13: Client Accounts Sub-Process Diagram.

Staff Salaries: Decision Table				
Outstanding Loan (OL)	**Loan Request (LR)**	**OL + LR**	**(OL + LR) ≤ 25% of Salary**	**Decision**
Yes	No	Yes	Yes	Deduct repayment
Yes	Yes	Yes	Yes	Grant Loan
Yes	Yes	Yes	No	Refuse Loan
No	No	No	-	No Deduction

Table 5.3: Staff Salaries: Decision Table

Calculate PRP: Decision Table				
Outstanding (Absence ≤ 3)	**Okay (Absence = [4|5])**	**Underachieved (Absence (= [6|7])**	**Critical (Absence > 7)**	**Decision**
Yes	No	No	No	3% increment
No	Yes	No	No	2% increment
No	No	Yes	No	1% increment (Warning)
No	No	No	Yes	No Increment Disciplinary Action

Table 5.4: Calculate PRP: Decision Table

Redmound-IBM checks the requirement that administrative overheads be covered by a monthly petty cash imprest of £100. Hence, we assume the *Overheads* sub-process is triggered once every 30 days. We also model the requirement that the sub-process executes only if some event(s) cause(s) the imprest account to be less than £100, as depicted in Figure 5.15.

Figure 5.16 illustrates the *Training and Management* Sub-Process of the *Administration* participant. This sub-process follows one of two possible paths. The first path records employee attendance, logs employee absence and then sends a message to the sub-process end. The other path ensures a training course is offered to every member of staff at least once a year, and on completion of course, sends a message to sub-process end. However

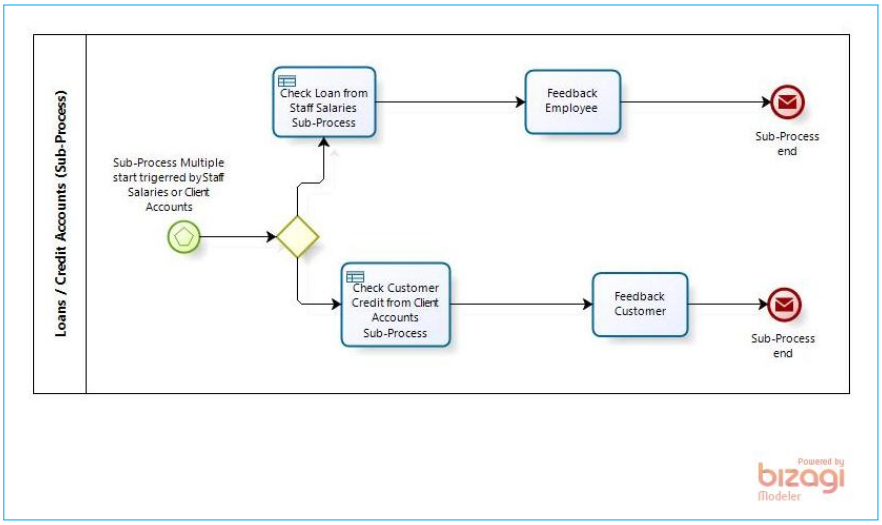


Figure 5.14: Loans / Credit Accounts Sub-Process Diagram.

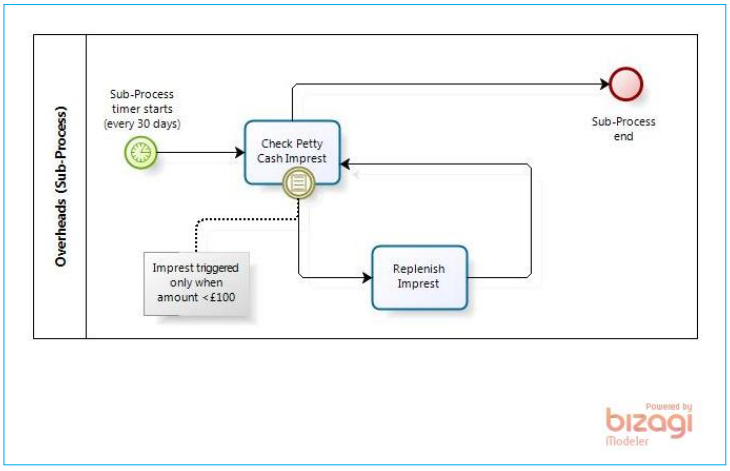


Figure 5.15: Overheads Sub-Process Diagram.

Request Credit: Decision Table				
Outstanding Credit (OC)	**Credit Request (CR)**	**OC + CR**	**(OC + CR) ≤ £100 and days credit taken ≤ 30**	**Decision**
Yes	Yes	Yes	Yes	Grant Credit
No	Yes	Yes	Yes	Grant Credit
Yes	Yes	Yes	No	Refuse Credit, add 30% interest and Blacklist.
No	Yes	Yes	No	... Ditto ...

Table 5.5: Request Credit: Decision Table

Check Loan / Credit Accounts: Decision Table		
Check Loan OK	**Check Credit OK**	**Decision**
Yes	-	Notify employee
No	-	Advise employee
-	Yes	Notify customer
-	No	Advise customer

Table 5.6: Check Loan / Credit Accounts: Decision Table

because more than one course may be taken within a year, there is a loop-back to the *Offer Training* service task. We elect to use a service task to automate the offer of available training options here to free users for other important business tasks. But either way, our information flow analysis is not affected.

Finally, we illustrate the *Personnel Records* sub-process in Figure 5.17. Notice that there are three possible flow paths here. One simply records the employee name, assigns an employee ID, stores records in *Employee Records dB* and terminates. Another path, taken on request, uses the business rule task *Monitor Employee Status* to check whether a person is an employee or not. This yields two paths, the first of which removes (former) employee's

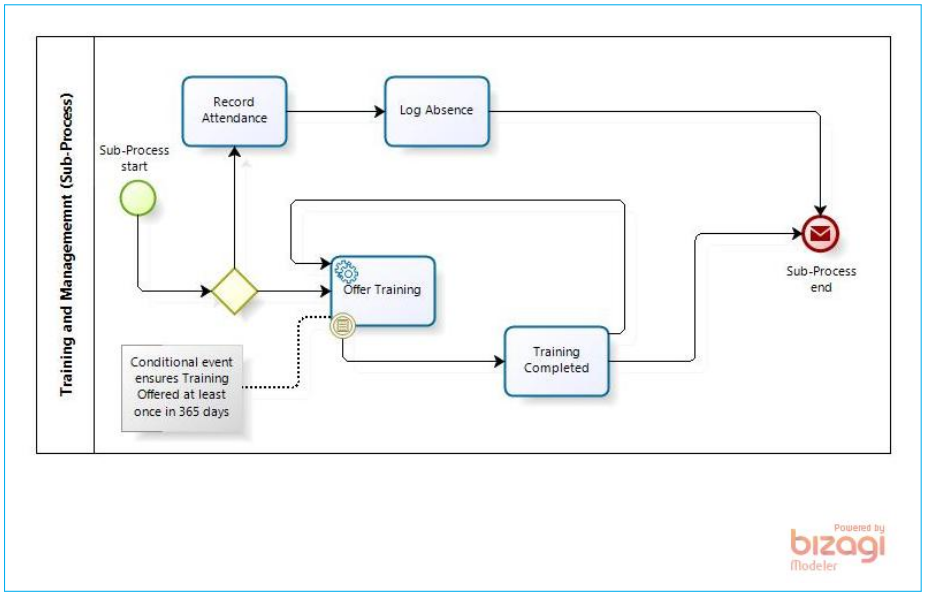


Figure 5.16: Training and Management Sub-Process Diagram.

records from the database (in the case where employee resigns or is sacked) and sends feedback to Sub-Process end. The other path only sends feedback to Sub-Process end in the case where the person is confirmed as a current employee.

We summarise in Table 5.7 the decision logic used to determine the flow path by the business rule task *Monitor Employee Status*.

Monitor Employee Status: Decision Table			
Employee ID	**Employed?**	**Employee Records dB**	**Decision**
Yes	No	Remove Records	Negative
Yes	Yes	-	Positive
No	No	No	Negative

Table 5.7: Monitor Employee Status: Decision Table

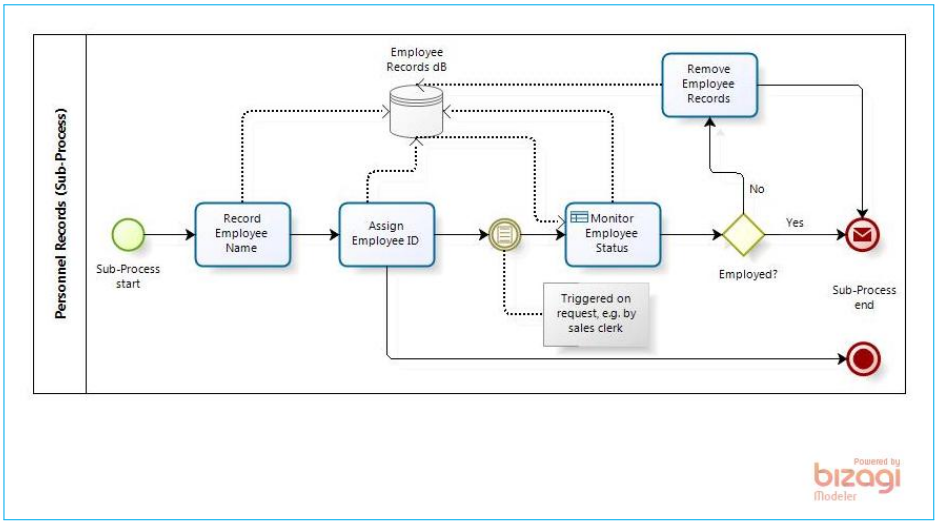


Figure 5.17: Personnel Records Sub-Process Diagram.

5.2.4.3 System Catalogue

In this section, we present the system catalogue (aka. *Data Dictionary*) which is basically a precise and organised repository (or catalogue) of the metadata pertaining to all the components and objects defined in the system [133], [144], [147], [141], [143]. The system catalogue comprise the inputs, outputs, intermediate calculations and data storage objects defined in the context of Redmound-IBM. We use the common notation described in Table 5.8 in our system catalogue:

We group the data elements defined in our system catalogue according to the process wherein they are defined, as illustrated in Figure 5.18.

Redmound-IBM:

Redmound-IBM = SelectProduct + ManageStock + Ts_ProductRelease+
StaffSalaries + ClientAccounts + Loan_CredAccounts +
Overheads + Training_Mgt + PersonnelRecords
IT Reseller Application

Select Product:

SelectProduct = productID + selectedProduct +
altProduct + suggestAlt +

System Catalogue Notation	
Notation	**Meaning**
=	"is defined as" or "is composed of" or "means"
+	AND
()	optional, i.e., "occurs 0 or 1 time"
{}	iteration, i.e., "may or may not include"
[]	selection, i.e., "select one of several choices"
* *	comment enclosure; **, with no text, means null comment
|	OR, i.e., partitions choices in []

Table 5.8: System Catalogue Notation

selectOptions + optionSelected
components and data elements in process
productID = VALID_PID
VALID_PID = 1{[0 – 9]}5 *range: 55550 – 99999*
valid product identifiers in catalogue
selectedProduct = VALID_PID
altProduct = VALID_PID
suggestAlt = *checks product list and suggests alternative*
OPTIONS = {Red, Blue, Green, Black, Pink}
selectOptions = *prompts customer to choose options*
optionSelected = VALID_PID
po = VALID_PID *output variable*
op = OPTIONS *output variable*
ss = VALID_PID *output variable*

Manage Stock:

ManageStock = PRODUCT_LIST + stockID +
OPTION_LIST + checkStockList + checkVendor +
discontinued + updateStockList +
price + priceList + updatePriceList

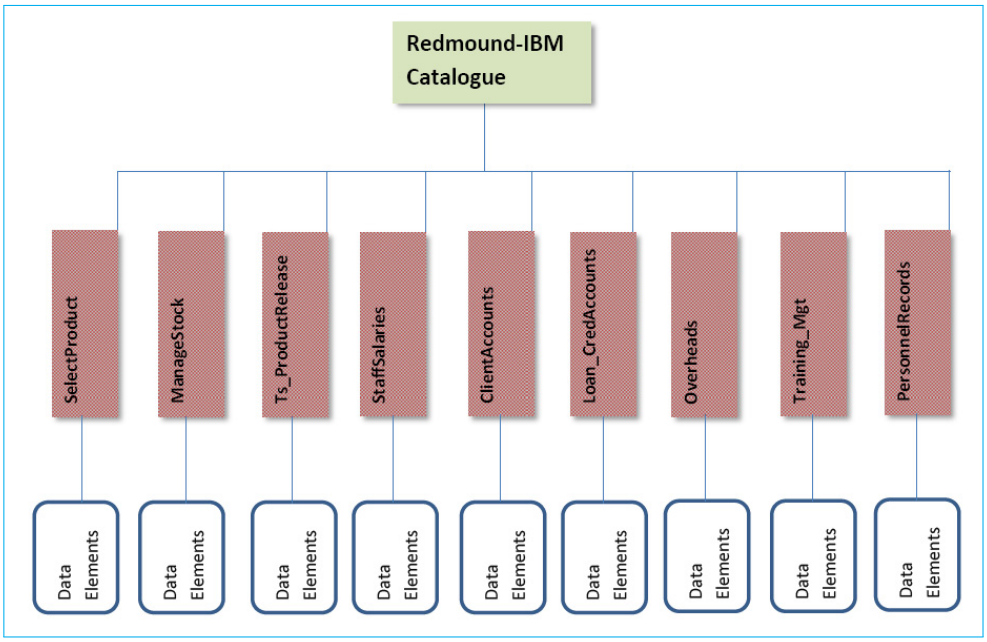


Figure 5.18: Redmound-IBM System Catalogue Structure.

PRODUCT_LIST = {VALID_PID} *Lists all products on the market*
stockID = VALID_SID
VALID_SID = 1{[0 – 9]}5 *range: 55550 – 99999*
stockList = {VALID_SID} *lists products held in stock*
OPTION_LIST = {VALID_OPTIONS} *lists available options*
VALID_OPTIONS = [*Red* | *Blue* | *Green* | *Black* | *Pink*]
checkStockList = *checks product in local productList*
checkVendor = *checks product with vendors*
discontinued = * boolean: 0 = false, and 1 = true values: [0 | 1]*
updateStockList = *removes discontinued products from productList*
price = ** *units: £ range: 1 – 10000 precision: nearest £*
priceList = { (+ stockID + , + price +) }
updatePriceList = *updates priceList as necessary*
bb = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
cc = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
vv = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
ww = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*

Transaction and Product Release:

Ts_ProductRelease = confirmPayment + releaseByShipping +
shippingCost + addPaperwork
confirmPayment = * boolean: 0 = false, and 1 = true values: [0 | 1]*
releaseByShipping = * boolean: 0 = false, and 1 = true values: [0 | 1]*
shippingCost = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999 *
precision: nearest £
addPaperwork = *ensures receipt and delivery note added*
values: [0 | 1]
pr = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
uu = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*

Staff Salaries:

StaffSalaries = loanStatus + outstandingLoan + requestedLoan
updatedLoan + grantLoan + repaymentAmount +
salaryB4Deduction + salaryAfter
loanStatus = * boolean: 0 = clear, and 1 = in arrears values: [0 | 1]*
outstandingLoan = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999*
precision: nearest £
requestedLoan = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999*
precision: nearest £
updatedLoan = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999*
precision: nearest £
grantLoan = * boolean: 0 = false, and 1 = true values: [0 | 1]*
repaymentAmount = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999*
precision: nearest £
salaryB4Deduction = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999*
precision: nearest £
salaryAfter = 1{ [0 – 9] }3 ** *units: £ range: 0 – 999*
precision: nearest £
ee = 1{[0 – 9]}6 *output* *Range: 111100 - 111199*
gg = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*

Performance Related Pay:

PerfRelatedPay = calculatePRP *calls operation calculatePRP*

calculatePRP = *calculates PRP* *input: employeeID*
output units: £ range: 0 - 999
outstanding = [0 – 3] *days absent* *units: days range: 0 – 3*
okay = [4 | 5] *days absent* *units: days range: 4 – 5*
underachieved = [6 | 7] *days absent* *units: days range: 6 – 7*
critical = 1{ [0 – 9] }3 *days absent* *units: days range: 8 – 365*
issueWarning = * boolean: 0 = false, and 1 = true values: [0 | 1]*
takeDisciplinaryAction = * boolean: 0 = false, and 1 = true*
values: [0 | 1]
pp = 1{[0 – 9]}3 *output* Range: 0 - 200*

Client Accounts:

ClientAccounts = registered + premiumAccount
manages client accounts
registered = * boolean: 0 = false, and 1 = true values: [0 | 1]*
purchaseTarget = 1{ [0 – 9] }4 ** *units: £ range: 0 – 9999*
targetMet = * boolean: 0 = false, and 1 = true values: [0 | 1]*
premiumAccount = * boolean: 0 = false, and 1 = true values: [0 | 1]*
requestCredit = * boolean: 0 = false, and 1 = true values: [0 | 1]*
grantCredit = * boolean: 0 = false, and 1 = true values: [0 | 1]*
outstandingCredit = 1{ [0 – 9] }3 ** *units: £ range: 0 – 100*
interest = 1{ [0 – 9] }3 ** *units: £ range: 0 – 100*
addToBlacklist = *adds customerID to bad creditors' list*
input: customerID output units: list
customerID = 1{ [0 – 9] }6 ** *range: 330001 – 339999*
tt = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
cc = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
rr = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*
bb = *output* * boolean: 0 = false, and 1 = true values: [0 | 1]*
aa = VALID_PID *output variable*

Loan / Credit Accounts:

Loan_CredAccounts = [checkLoanAccount | checkCreditAccount]
checkLoanAccount = *checks status of employee loan account*
checkCreditAccount = *checks status of customer credit account*
employeeFeedback = ** *values: [0 | 1]*

customerFeedback = ** *values: [0 | 1]*
ll = *output* * boolean: 0 = false, and 1 = true values: [0 | 1]*
mm = 1{[0 – 9]}3 *output* *Range: 0 - 999*
dd = 1{[0 – 9]}3 *output* *Range: 0 - 999*
xx = 1{[0 – 9]}6 *output* *Range: 111100 - 111199*
yy = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*

Overheads:

Overheads = **
pettyCash = 1{[0 – 9]}3 *holds monthly petty cash imprest*
units: £ range: 0 – 100
monthCounter = 1{[0 – 9]}2 *counts up to 30* *range: 1 – 30*
balance = 1{[0 – 9]}3 *units: £ range: 0 – 100*
updatePettyCash = *updates petty cash imprest to £100*

Training and Management:

Training_Mgt = **
attendanceRecords = 1{[0 – 9]}3 *units: days range: 0 – 240*
absence = 1{[0 – 9]}3 *units: days range: 0 – 240*
coursesCompleted = [0 – 9] *range: 0-5*
offerTraining = * boolean: 0 = false, and 1 = true values: [0 | 1]*
getCoursesCompleted = *returns number of courses completed*
[0 – 9] *range: 0-5*
getAbsenceRecords = *returns absence in business year*
1{[0 – 9]}3 *units: days range: 0 – 240*
maxAttendance = 1{[0 – 9]}3 * parameter* *units: days
range: 0 – 240*
maxCourses = [0 – 9] * parameter* *range: 0-5*
yearCounter = 1{[0 – 9]}3 *units: days range: 0 – 365*
ii = 1{[0 – 9]}6 *output* *Range: 111100 - 111199*
jj = 1{[0 – 9]}3 *output* *Range: 0 - 240*
kk = 1{[0 – 9]}6 *output* *Range: 111100 - 111199*
qq = [0-5] *output* *Range: 0 - 5*
ff = 1{[0 – 9]}6 *output* *Range: 111100 - 111199*
hh = 1{[0 – 9]}3 *output* *Range: 0 - 240*
nn = 1{[0 – 9]}6 *output* *Range: 111100 - 111199*

mz = [0-5] *output* *Range: 0 - 5*

Personnel Records:

PersonnelRecords = *registers new employee and updates employee records*
employeeName = **
employeeID = 1{ [0 – 9] }6 ** *range: 111101 – 111199*
isEmployee = * boolean: 0 = false, and 1 = true values: [0 | 1]*
removeEmployee = *removes employee records* *input: employeeID*
employeeStatus = * boolean: 0 = false, and 1 = true values: [0 | 1]*
zz = *output* *boolean: 0 = false, and 1 = true values: [0 | 1]*

For ease of reference, especially with respect to the security classification of system variables, which we will present in Section 5.3, we tabulate the technical elements defined in Redmound-IBM is Table 5.9. Notice the table is organised by processes, which we later develop into B machines. Variable names (e.g., parameters) that appear in more than one process (or machine) have the same type and are defined only at first occurrence in the table. Notice also that we list the output variables separately at the end of the table. This is to draw attention to the variables we need to check with our information flow analyser to ensure no variable(s) with a higher security classification flow(s) into any output variable with a lower security classification.

Table 5.9: Technical Elements and Types.

Technical Elements

SelectProduct

Element Name	**Element Type**	**Comments**
productID	Variable	Identifier. Value: 55550 - 99999
selectedProduct	Variable	Identifier. Value: 55550 - 99999
altProduct	Variable	Identifier. Value: 55550 - 99999
optionSelected	Variable	$[Red \mid Blue \mid Green \mid Black \mid Pink]$
VALID_PID	Set	Set of Values: 55550 - 99999

Continued on next page

Table 5.9 – continued from previous page		
Element Name	**Element Type**	**Comments**
OPTIONS	Set	$\{Red,\ Blue,\ Green,\ Black,\ Pink\}$
prod	Parameter	Element of Values: 55550 - 99999
option	Parameter	$[Red \mid Blue \mid Green \mid Black \mid Pink]$
sAlt	Parameter	Element of Values: 55550 - 99999
color	Parameter	$[Red \mid Blue \mid Green \mid Black \mid Pink]$

Vendors

Element Name	**Element Type**	**Comments**
PRODUCTS	Set	Deferred set of products
PRICE	Set	Element of 1..10000
products	Variable	Subset of PRODUCTS
salePrice	Variable	Element of PRICE
stocked	Variable	Value: $products \rightarrow PRICE$
prc	Parameter	Element of PRICE
item	Parameter	Element of PRODUCT

ManageStock

Element Name	**Element Type**	**Comments**
OPTION_LIST	Set	$\{Red,\ Blue,\ Green,\ Black,\ Pink\}$
VALID_SID	Set	Set of Values: 55550 - 99999
PRICE_RANGE	Set	Set of Values: 1 - 10000
stockID	Variable	Element of Values: 55550 - 99999
stockList	Variable	Element of Values: 55550 - 99999
discontinued	Variable	Element of **BOOL**
price	Variable	Set of Values: 1 - 10000
priceList	Variable	Value: $stockList \rightarrow PRICE\_RANGE$
product	Parameter	Element of VALID_SID
option	Parameter	Element of OPTION_LIST

ClientAccounts

Element Name	**Element Type**	**Comments**
CUSTOMERID	Set	Set of Values 330001 - 339999

Continued on next page

Table 5.9 – continued from previous page		
Element Name	**Element Type**	**Comments**
default_CustID	Constant	Value: 330001
purchaseTarget	Constant	Value: 5000
maxCredit	Constant	Value: 100
maxDays	Constant	Value: 30
registered	Variable	Subset of CUSTOMERID
customerID	Variable	Element of CUSTOMERID
premiumAccount	Variable	Element of **BOOL**
outstandingCredit	Variable	Element of **NAT**
interest	Variable	Element of **NAT**
grantCredit	Variable	Element of **BOOL**
custID	Parameter	Element of CUSTOMERID
purchase	Parameter	Element of **NAT**
dy	Parameter	Element of **NAT**
cId	Parameter	Element of CUSTOMERID
amount	Parameter	Element of **NAT**

Customer

Element Name	**Element Type**	**Comments**
ITEMID	Set	Set of Values: 55550 - 99999
wares	Variable	Subset of ITEMID
return	Variable	Element of **BOOL**
returnItem	Variable	Element of *wares*
faulty	Variable	Value: $wares \rightarrow$ **BOOL**

PersonnelRecords

Element Name	**Element Type**	**Comments**
EMP_ID_RANGE	Set	Set of Values 111100 - 111199
employees	Set	Subset of EMP_ID_RANGE
employeeName	Variable	Element of **BOOL**
employeeID	Variable	Element of EMP_ID_RANGE
employeeStatus	Variable	Element of **BOOL**
empID	Parameter	Element of EMP_ID_RANGE

Continued on next page

Table 5.9 – continued from previous page		
Element Name	**Element Type**	**Comments**
maxAttendance	Parameter	Element of **NAT**
maxCourses	Parameter	Element of **NAT**
ANNUAL_ ATTENDANCE	Set	Values: 0 - 240
attendanceRecords	Variable	Element of Values: 0 - 240
absence	Variable	Element of **NAT**
coursesCompleted	Variable	Element of **NAT**
emp	Variables	Values: 111100 - 111199
offerTraining	Variables	Element of **BOOL**

PerfRelatedPay

Element Name	**Element Type**	**Comments**
abs	Parameter	Element of **NAT**
crsComp	Parameter	Element of **NAT**
outstanding	Variable	Element of **NAT**
okay	Variable	Element of **NAT**
underachieved	Variable	Element of **NAT**
critical	Variable	Element of **NAT**
warning	Variable	Element of **BOOL**
discipline	Variable	Element of **BOOL**
prp	Variable	Element of **NAT**
salary	Variable	Element of **NAT1**

StaffSalary

Element Name	**Element Type**	**Comments**
LOANSTATUS	Set	Values: $\{Owing,\ NotOwing\}$
EMPID	Set	Values: 111100 - 111199
identifier	Set	Subset of EMPID
loanStatus	Variable	Values : $identifier \rightarrow LOANSTATUS$
outstandingLoan	Variable	Element of **NAT**
requestedLoan	Variable	Element of **NAT**

Continued on next page

Table 5.9 – continued from previous page		
Element Name	**Element Type**	**Comments**
repaymentAmount	Variable	Element of **NAT**
salaryB4Deduction	Variable	Element of **NAT**
salaryAfter	Variable	Element of **NAT**
grantLoan	Variable	Element of **BOOL**

Loan_CredAccounts

Element Name	Element Type	Comments
loanSts	Variable	Element of LOANSTATUS
creditStatus	Variable	Element of **NAT**

Accounts

Element Name	Element Type	Comments
GUESTACCOUNTS	Set	$\{ga11001,\ ga11002,\ ga11003,$ $ga11004,\ ga11005\}$
guestAccount	Variable	Element of GUESTACCOUNTS
guestID	Variable	Element of GUESTACCOUNTS

Ts_ProductRelease

Element Name	Element Type	Comments
paymentConfirmed	Variable	Element of **BOOL**
toShip	Variable	Element of **BOOL**
weight	Variable	Element of **NAT1**
shippingCost	Variable	Element of **NAT**
documentsAdded	Variable	Element of **BOOL**

Sales_Technical

Element Name	Element Type	Comments
pID	Variable	Element of **NAT1**
stockState	Variable	Element of **BOOL**

Continued on next page

Table 5.9 – continued from previous page		
Element Name	**Element Type**	**Comments**

Overheads

Element Name	Element Type	Comments
limit	Constant	Values: 30
monthCounter	Variable	Values: ≤ limit
balance	Variable	Element of **NAT**
pettyCash	Variable	Values: ≤ PETTYCASH

Administration

Element Name	Element Type	Comments
counter	Parameter	Element of **NAT1**

Output Variables

Element Name	Element Type	Comments
aa	Variable	Element of VALID_PID
bb	Variable	Element of **BOOL**
cc	Variable	Element of **BOOL**
dd	Variable	Element of **NAT**
ee	Variable	Values: 111100 - 111199
ff	Variable	Values: 111100 - 111199
gg	Variable	Element of **BOOL**
hh	Variable	Values: 0 - 240
ii	Variable	Values: 111100 - 111199
jj	Variable	Values: 0 - 240
kk	Variable	Values: 111100 - 111199
ll	Variable	Element of **BOOL**
mm	Variable	Element of **NAT**
nn	Variable	Values: 111100 - 111199
pp	Variable	Element of **NAT**
qq	Variable	Values: 0 - 5
rr	Variable	Element of **BOOL**
ss	Variable	Element of VALID_PID

Continued on next page

Table 5.9 – continued from previous page		
Element Name	**Element Type**	**Comments**
tt	Variable	Element of **BOOL**
uu	Variable	Element of **BOOL**
vv	Variable	Element of **BOOL**
ww	Variable	Element of **BOOL**
xx	Variable	Values: 111100 - 111199
yy	Variable	Element of **BOOL**
zz	Variable	Element of **BOOL**
po	Variable	Element of VALID_PID
op	Variable	Element of OPTIONS
pr	Variable	Element of **BOOL**
mz	Variable	Values: 0 - 5

Following from our system analysis in Sections 5.2.4.1 and 5.2.4.2, we summarise, in Section 5.2.5, the visibility relationships arising from the structural relationship between the processes and sub-process in Redmound-IBM.

5.2.5 Redmound-IBM: The System Synthesis

In this section, we present the visibility relationships between the processes and sub-processes introduced in Sections 5.2.4.1 and 5.2.4.2, which forms the bases for the B Machine subsystems we use later in our formal specification in Section 5.4. We begin with a text listing of the visibility relationships.

- ManageStock SEES Vendors
- Customer SEES SelectProduct
- Training_Mgt SEES PersonnelRecords
- PerfRelatedPay SEES PersonnelRecords, Training_Mgt
- StaffSalaries SEES Training_Mgt, PersonnelRecords, PerfRelatedPay
- Loan_CredAccounts SEES StaffSalaries, ClientAccounts

- Accounts EXTENDS StaffSalaries, ClientAccounts, Loan_CredAccounts
- Ts_ProductRelease SEES Customer, Loan_CredAccounts, ManageStock
- Sales_Technical INCLUDES ManageStock, Ts_ProductRelease
- Administration INCLUDES Overheads, Training_Mgt, PersonnelRecords

We further elaborate on the structure of the formal B specification of our Redmound-IBM reseller monitor in Figure 5.19. In this figure we use rounded rectangles to represent B Machine subsystems within Redmound-IBM. We use labeled directed lines between two machines to indicate visibility (structuring) links between connected machines via B structuring mechanisms, with the arrow pointing at the machine that has visibility of the other connected machine. With respect to our information flow analysis, the arrow in effect points at the machine into which information may flow from the static parts of connected (visible) machine(s) by virtue of structuring mechanisms in B. The label on a directed line indicates the type of visibility rule that exists between the connected machines.

Using the structural relationships described in this section, we present the formal specification and design of the B Machine subsystems used to build Redmound-IBM IT Reseller Monitor in Section 5.4.1. In the following section, though, we introduce our definition of the information flow policy defined on the variables in the development.

5.3 Information Flow Policy

In this section, we present the information flow policy definitions for Redmound-IBM. The definitions given here are used in subsequent sections for the flow analysis of the machines, refinements and implementations.

5.3.1 Redmound-IBM: Policy Definitions

We specify in this section the information flow policy for Redmound-IBM. The variables are mapped to security classes on the security flow lattice de-

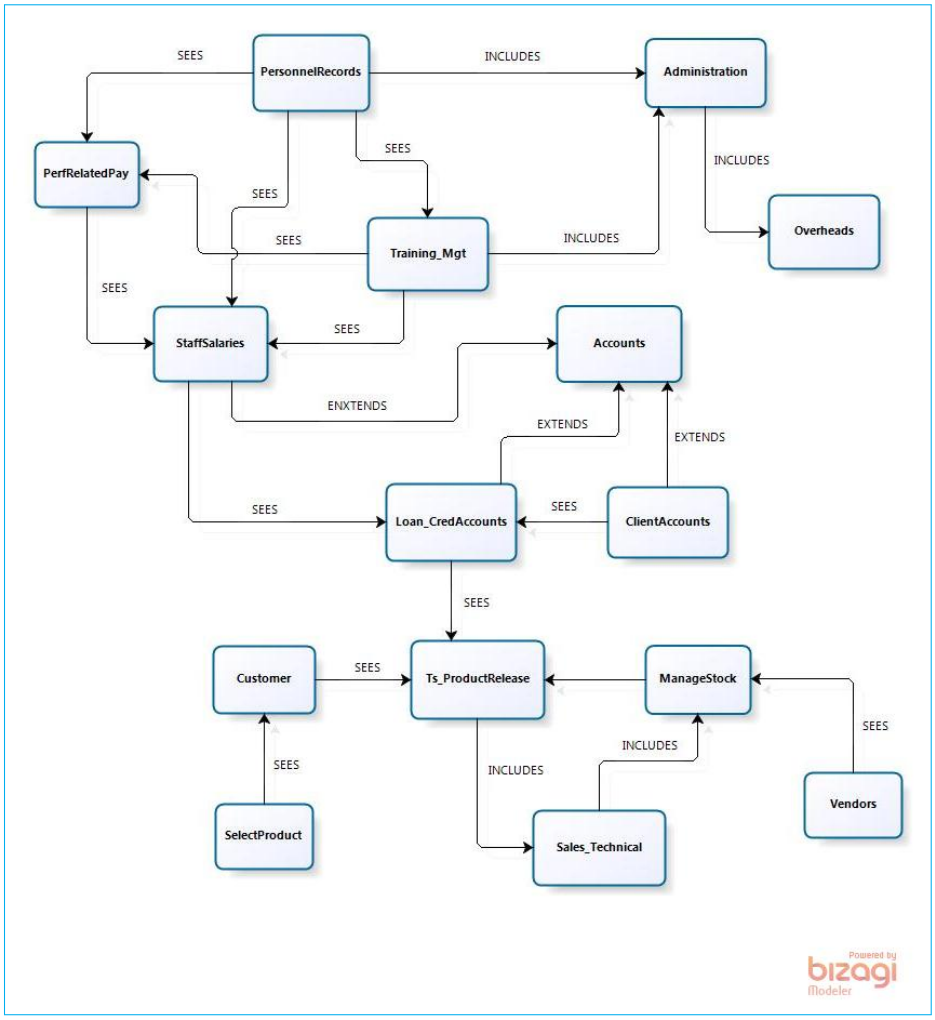


Figure 5.19: Structure of B Machine Specifications

picted in Figure 5.20, where the security classes are denoted ∅, [**A**], [**B**], [**C**], [**AB**], [**AC**], and [**ABC**]. The representation is purely syntactic; it is intended only to provide a visual image of secure information flow, for example, variables mapped to [**A**] could be seen to *securely* flow into variables mapped to [**AB**]. The greatest lower bound of all elements of the security flow lattice is ∅, whereas the least upper bound is [**ABC**]. We employ the following notation, with variables grouped in sets by security classification:

$\underline{\varnothing}$ denotes the set $\{productID,\ selectedProduct,\ altProduct,$
$optionSelected,\ prod,\ option,\ sAlt,\ color,\ po,\ op,$

ss, $products$, $item$, $stockList$, $stockID$, pId,
$discontinued$, $price$, $PRODUCT\_LIST$, $weight$,
$OPTION\_LIST$, cc, $paymentConfirmed$, $toShip$,
$shippingCost$, $documentAdded$, $VALID\_PID$, },
i.e., the set of *public* variables with security classification ∅

A denotes {$salePrice$, $stocked$, prc, bb, $PRICE\_RANGE$, SID,
$product$, zz},
i.e., the set of variables with security classification [**A**], or the
domain of the set of variable mappings to the security class [**A**].

B denotes {$CUSTOMERID$, $purchaseTarget$, $maxCredit$, $maxDays$,
$registered$, $customerID$, $custID$, $purchase$, tt, $OPTIONS$,
cId, dy, $amount$, $wares$, $faulty$, $returnItem$, $PRODUCTS$,
$return$, $GUESTACCOUNTS$, $guestAccount$, dd},
i.e., the set of variables with security classification [**B**].

C denotes {EMP_ID_RANGE, $employeeName$, $employeeStatus$,
$empID$, $ANNUAL\_ATTENDANCE$, emp,
$maxCourses$, abs, jj, $identifier$, $LOANSTATUS$, ee, gg,
$maxAttendance$, $employees$}
i.e., the set of variables with security classification [**C**].

AB denotes {$default\_CustID$, $premiumAccount$, $outstandingCredit$,
$interest$, $grantCredit$, rr, $creditStatus$, mm,
$stockState$, vv, ww, uu},
i.e., the set of variables with security classification [**AB**].

AC denotes {$attendanceRecords$, $absence$, $coursesCompleted$,
$offerTraining$, ii, kk, qq, $crsComp$, $outstanding$, $okay$,
$underachieved$, $critical$, $warning$, $discipline$, prp, pp},
i.e., the set of variables with security classification [**AC**]

ABC denotes {$salary$, $loanStatus$, $outstandingLoan$, $requestedLoan$,
$repaymentAmount$, $salaryB4Deduction$, $salaryAfter$,
$grantLoan$, $loanSts$, ll, $limit$, $PETTYCASH$, $pettyCash$,

balance, *monthCounter*, *counter*, *xx*},
i.e., the set of variables with security classification [**ABC**].

Each of the variables in the above sets is pointwise mapped to the corresponding security class, for example each of the variables in **AB** maps to the security class [**AB**]. We illustrate this by writing $\underline{AB} \longrightarrow$ [**AB**] in the security flow lattice shown in Figure 5.20. Furthermore, we write:

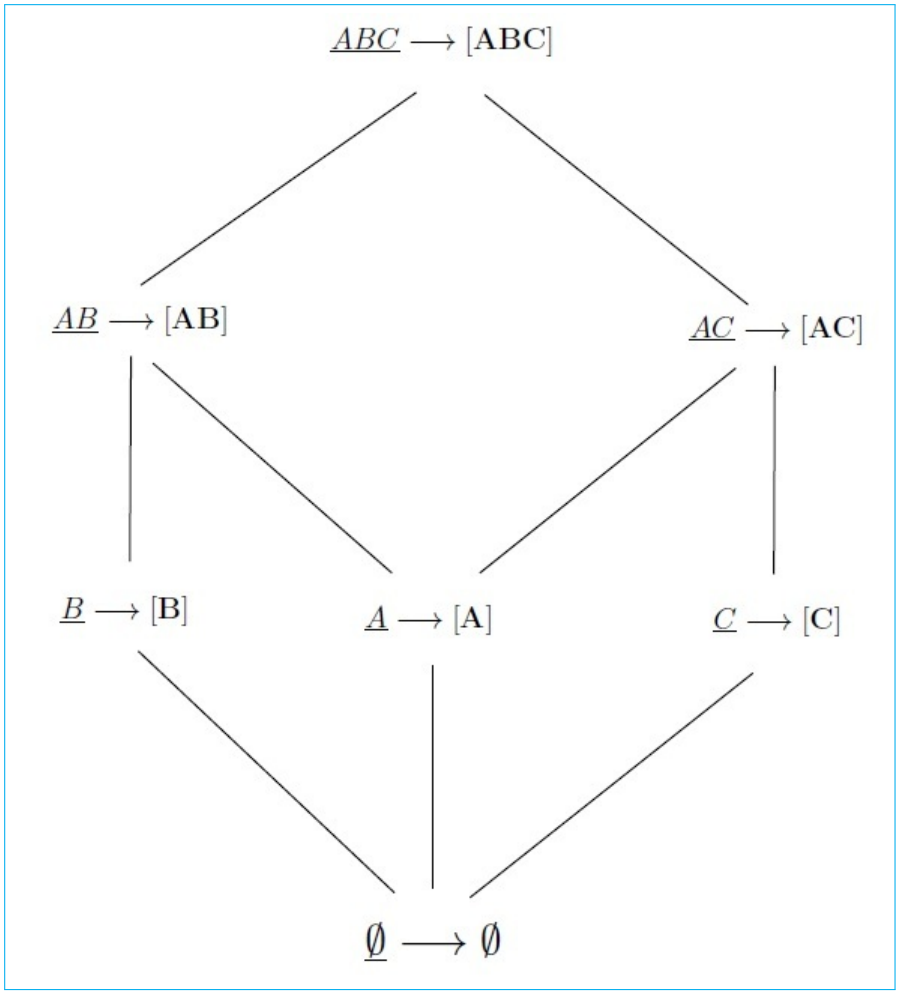


Figure 5.20: Redmound-IBM Security Policy Lattice

$\downarrow$**A** to denote the set of variables mapped to security classifications lower than or equal to [**A**] on the security flow lattice (sfl).

$\downarrow$**B** to denote the set of variables mapped to security classifications lower than or equal to [**B**] on the sfl.

$\downarrow$**C** to denote the set of variables mapped to security classifications lower than or equal to [**C**] on the sfl.

$\downarrow$**AB** to denote the set of variables mapped to security classifications lower than or equal to [**AB**] on the sfl.

$\downarrow$**AC** to denote the set of variables mapped to security classifications lower than or equal to [**AC**] on the sfl.

$\downarrow$**ABC** to denote the set of variables mapped to security classifications lower than or equal to [**ABC**] on the sfl.

Using the notation given above, we itemise our defined pointwise security mappings in terms of what information may be read by individual variables.

- **A** may read $\downarrow$**A**;
- **B** may read $\downarrow$**B**;
- **C** may read $\downarrow$**C**;
- **AB** may read $\downarrow$**AB**;
- **BC** may read $\downarrow$**BC**;
- **ABC** may read $\downarrow$**ABC**;
- All variables may read $\varnothing$.

In the following sections, we use the security definition here to check Redmound-IBM for secure information flow. First, we introduce the formal B machines used to model Redmound-IBM in Section 5.4.

5.4 Redmound-IBM: The B Components

Following the structured development approach illustrated in Figure 5.19, we present in this section the formal B Machine specifications of the subsystems used in our development of Redmound IT reseller Business Monitor (aka Redmound-IBM). Following the development approach in the B Method, we structure this section by Machines (Specifications), Refinements and Implementations. We then apply our information flow analyser to each

machine, refinement and implementation to check whether flows between system variables respect the security flow lattice defined in Figure 5.20.

5.4.1 Redmound-IBM: Formal Specification and Design

In this subsection, we present the B Machines used in our development of Redmound-IBM. All the B Machines discussed here have been checked and proved for proof obligation consistency using Atelier B. A screenshot of the Redmound-IBM project in Atelier B showing the proof obligations automatically generated and proved is given in Figure 5.21.

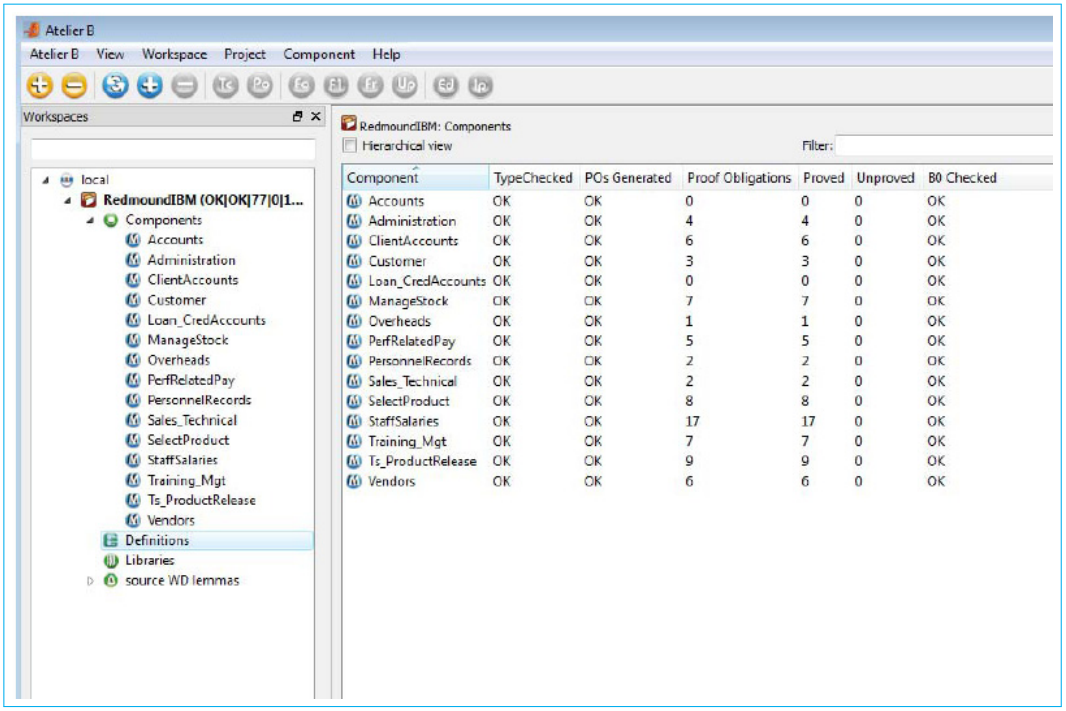


Figure 5.21: Redmound-IBM: Proof Obligations Consistency Check

We discuss, first, the *SelectProduct* machine which is visible to the *Customer* machine via the **SEES** structuring mechanism. We include in the **INVARIANT** clause the non-trivial requirement that *productID*, *altProduct*, and *selectedProduct* must not accept any natural number that is not a valid identifier. The *SelectProduct* machine presents three client interfaces

through the following operations:

- $po \longleftarrow selectProduct(prod)$: This operation takes as input data of type *VALID_PID*, updates the *selectedProduct* state variable and outputs the value on *po*.
- $op \longleftarrow selectOptions(option)$: This operation updates the *optionSelected* state variable with the input passed via the operation parameter, and outputs the value on *op*.
- $ss \longleftarrow suggestAlternative(sAlt, color)$: This operation takes two parameters of types *VALID_PID* and *OPTIONS* respectively, updates the *productID*, *altProduct* and *optionSelected* state variables, and outputs the first parameter on *ss*.

The MACHINE *SelectProduct*, with its three interfaces is illustrated in Figure 5.22.

Figure 5.22: Redmound-IBM: MACHINE SelectProduct

The following formal listing is the B Machine component *SelectProduct*, which defines the client interfaces described above.

MACHINE $SelectProduct$

SETS $OPTIONS = \{Red, Blue, Green, Black, Pink\}$

PROPERTIES $OPTIONS \neq \varnothing$

DEFINITIONS $VALID\_PID \triangleq 55550..99999$

VARIABLES

$productID,\ selectedProduct,\ altProduct,\ optionSelected$

INVARIANT

$productID \in VALID\_PID \wedge selectedProduct \in VALID\_PID \wedge$
$altProduct \in VALID\_PID \wedge optionSelected \in OPTIONS \wedge$
$\forall pId \cdot (pId \in \mathbf{NAT} \ \wedge \ pId \notin VALID\_PID \Rightarrow$
$\quad pId \neq productID) \wedge$
$\forall altProd \cdot (altProd \in \mathbf{NAT} \ \wedge \ altProd \notin VALID\_PID \Rightarrow$
$\quad altProd \neq altProduct) \wedge$
$\forall selProd \cdot (selProd \in \mathbf{NAT} \ \wedge \ selProd \notin VALID\_PID \Rightarrow$
$\quad selProd \neq selectedProduct)$

INITIALISATION $productID,\ selectedProduct,\ altProduct,$
$optionSelected \ := \ 55550,\ 55550,\ 99999,\ Red$

OPERATIONS

$po <-- selectProduct(prod) \ \triangleq$
PRE $prod : VALID\_PID$ **THEN**
$selectedProduct \ := \ prod \ ||$
$po \ := \ prod$
END;

$op <-- selectOptions(option) \ \triangleq$
PRE $option : OPTIONS$ **THEN**
$optionSelected \ := \ option \ \|$
$op \ := \ option$
END;

$ss <-- suggestAlternative(sAlt,\ color) \ \triangleq$
PRE $sAlt \in NAT1 \ \wedge \ sAlt \in VALID\_PID \wedge$
$color : OPTIONS$ **THEN**
$productID := sAlt \ \|$
$altProduct := sAlt \ \|$

$optionSelected := color \ \|$
$ss := sAlt$

END

END

We now present the formal B machine component *Vendors*, which is *seen* by the *ManageStock* machine. MACHINE *Vendors* defines the operations:

- $setProduct$: Sets product pricing for stocked items.
- $bb \longleftarrow itemStocked(item)$: Checks for items in stock, and returns **TRUE** if found.

We present below the listing of MACHINE *Vendors*, viz:

MACHINE $Vendors$
SETS
$PRODUCTS$
PROPERTIES
$PRODUCTS \neq \varnothing$
DEFINITIONS
$PRICE \triangleq 1..10000$
VARIABLES
$products,\ salePrice,\ stocked$
INVARIANT
$products \subseteq PRODUCTS \ \wedge salePrice \in PRICE \wedge$
$stocked \in products \longrightarrow PRICE$
INITIALISATION
$products,\ salePrice,\ stocked := \varnothing,\ 1,\ \varnothing$
OPERATIONS
$setProduct \triangleq$
ANY $prod,\ prc$ **WHERE**
$prod \in products \ \wedge \ prc \in PRICE$ **THEN**
$stocked(prod) := prc$
END;

$bb \longleftarrow itemStocked(item) \triangleq$
PRE $item \in products$ **THEN**
$bb := \mathbf{TRUE}$
END
END

We discuss next the *ManageStock* machine, which *sees* MACHINE *Vendors*. We define in the **INVARIANT** clause the added requirement that any natural number used as *stockID* must be a valid identifier (i.e., value between 55550 and 99999). MACHINE *ManageStock* has four interfaces, namely:

- $bb \longleftarrow checkStockList(product, option)$: Checks for availability of product in local stock list. The operation returns **TRUE** if product is in stock, otherwise, it returns **FALSE**.
- $cc \longleftarrow checkVendor(product, option)$: Checks product list with vendors and returns **TRUE** if in stock, otherwise, it returns **FALSE** and sets the *discontinued* flag to **TRUE**.
- $updateStockList(prod)$: The operation does some housekeeping by removing discontinued products from local stock list.
- $updatePriceList(prod, prc)$: Updates the price list when called.

Both machines *ManageStock* and *Vendors* are depicted in Figure 5.23

The following listing shows the B Machine component *ManageStock*, viz:

MACHINE $ManageStock$
SEES $Vendors$
SETS $OPTION\_LIST = \{Red, Blue, Green, Black, Pink\}$
PROPERTIES $OPTION\_LIST \neq \varnothing$
DEFINITIONS $VALID\_SID \triangleq 55550..99999$;
$PRODUCT\_LIST \triangleq 55550..99999$;
$PRICE\_RANGE \triangleq 1..10000$
VARIABLES $stockID$, $stockList$, $discontinued$, $price$, $priceList$

INVARIANT $stockID \in VALID\_SID \wedge stockList \subseteq PRODUCT\_LIST \wedge$
$discontinued \in \mathbf{BOOL} \wedge price \in PRICE\_RANGE \wedge$

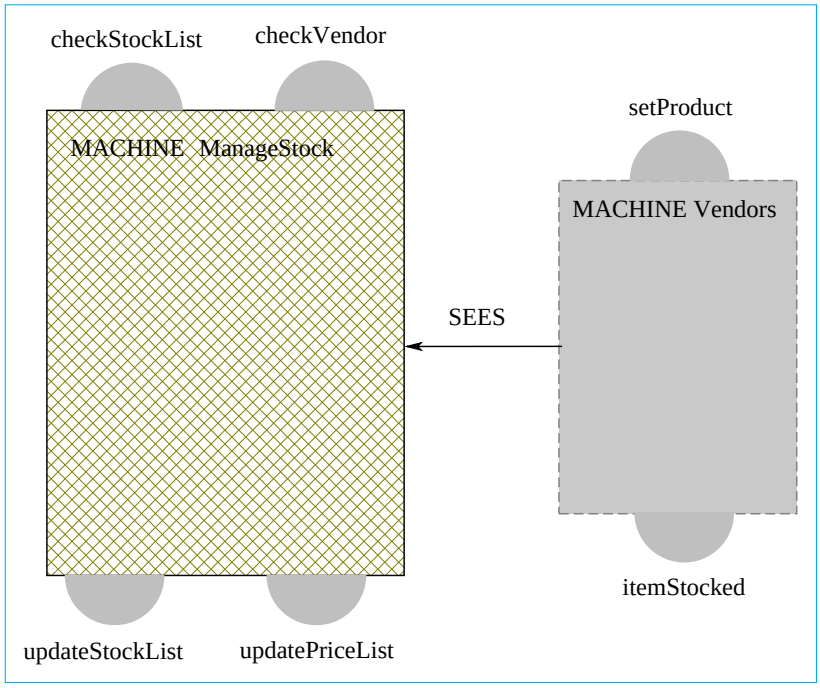


Figure 5.23: Redmound-IBM: MACHINE ManageStock

$$priceList \in stockList \longrightarrow PRICE\_RANGE \wedge$$
$$\forall sId \cdot (sId \in \mathbf{NAT} \wedge sId \notin VALID\_SID \Rightarrow sId \neq stockID)$$

INITIALISATION $stockID,\ stockList,\ discontinued,\ price,\ priceList$
$$:= 55550,\ 55550,\ FALSE,\ 1,\ 55550 \mapsto 1$$

OPERATIONS

$bb \longleftarrow checkStockList(product,\ option) \triangleq$
PRE $product \in VALID\_SID \wedge option \in OPTION\_LIST$
THEN
IF $product \in stockList$ **THEN**
$bb :=$ **TRUE**
ELSE
$bb :=$ **FALSE**
END
END;

$cc \longleftarrow checkVendor(product,\ option) \triangleq$
PRE $product \in VALID\_SID \wedge option \in OPTION\_LIST$

```
            THEN
                IF product ∈ PRODUCT_LIST THEN
                    cc := TRUE
                ELSE
                    discontinued := TRUE  ||
                    cc := FALSE
                END
            END;

        updateStockList(prod)  ≙
            PRE prod ∈ VALID_SID  ∧  prod ∈ stockList ∧
                prod ∉ PRODUCT_LIST THEN
                    stockList := stockList − {prod}
            END;

        updatePriceList(prod, prc)  ≙
            PRE prod ∈ VALID_SID  ∧  prod ∈ stockList ∧
                prc ∈ PRICE_RANGE THEN
                priceList(prod)  :=  prc
            END
    END
```

We present below the *ClientAccounts* machine, which has four interfaces, namely:

- $tt \longleftarrow targetMet(custID,\ purchase)$: This operation determines if the purchase target set is met by customers, whereupon such customer is upgraded to the *premiumAccount* status and thus eligible for customer credit allowance. The operation returns TRUE or FALSE to indicate whether target (specified in the CONSTANTS clause) is met or not.

- $setCustomerID(cId)$: This operation adds a *customerID* to the list of *registered* customers when called, after ensuring that input value is valid. It has no return value.

- $cc \longleftarrow getCustomerID(cId)$: This operation checks the list of *reg-*

istered customers for a specified *customerID* and returns TRUE if customer is registered; else, it returns FALSE.

- $rr \longleftarrow requestCredit(cId,\ amount)$: This operation checks that the business rules defined with respect to customer credit is met and returns TRUE or FALSE to indicate whether the customer is eligible or not.

The following is a listing of the B machine component *ClientAccounts*.

MACHINE *ClientAccounts*
DEFINITIONS
$CUSTOMERID \ \triangleq 330001..339999$
CONSTANTS
$default\_CustID,\ purchaseTarget,\ maxCredit,\ maxDays$
PROPERTIES
$CUSTOMERID \neq \varnothing \ \wedge \ default\_CustID = 330001 \ \wedge$
$purchaseTarget = 5000 \ \wedge \ maxCredit = 100 \ \wedge \ maxDays = 30$
VARIABLES
$registered,\ customerID,\ premiumAccount,$
$outstandingCredit,\ interest,\ grantCredit$
INVARIANT
$registered \subseteq CUSTOMERID \wedge customerID \in CUSTOMERID$
$\wedge\ premiumAccount \in \mathbf{BOOL} \wedge\ outstandingCredit : \mathbf{NAT} \ \wedge$
$interest \ \in \mathbf{NAT} \wedge\ grantCredit \in \mathbf{BOOL}$
INITIALISATION
$registered,\ customerID,\ premiumAccount,$
$outstandingCredit,\ interest,\ grantCredit \ :=$
$\varnothing,\ default\_CustID,\ FALSE,\ 0,\ 0,\ FALSE$
OPERATIONS
$tt \longleftarrow targetMet(custID,\ purchase) \ \triangleq$
PRE $custID \in CUSTOMERID \ \wedge \ purchase \in \mathbf{NAT} \ \wedge$
$purchase \leq 9999$ **THEN**
IF $purchase \geq purchaseTarget$ **THEN**
$premiumAccount := \mathbf{TRUE} \ \|$
$tt := \mathbf{TRUE}$
ELSE

$premiumAccount := \mathbf{FALSE}$ ||
$tt := \mathbf{FALSE}$
END
END;

$setCustomerID(cId)$ $\triangleq$
PRE $cId \in CUSTOMERID$ **THEN**
$registered := registered \cup \{cId\}$
END;

$cc <-- getCustomerID(cId)$ $\triangleq$
PRE $cId \in CUSTOMERID \ \wedge \ cId \notin registered$ **THEN**
IF $cId \in registered$ **THEN**
$cc := \mathbf{TRUE}$
ELSE
$cc := \mathbf{FALSE}$
END
END;

$rr \longleftarrow requestCredit(cId, \ amount)$ $\triangleq$
PRE $cId \in CUSTOMERID \ \wedge \ amount \in \mathbf{NAT} \ \wedge$
$amount + outstandingCredit \leq 100$ **THEN**
ANY dy **WHERE** $dy \in \mathbf{NAT}$ **THEN**
SELECT $dy < maxDays$ **THEN**
$outstandingCredit := outstandingCredit + amount$
|| $grantCredit := \mathbf{TRUE}$ ||
$rr := \mathbf{TRUE}$
WHEN $dy \geq maxDays \ \wedge$
$outstandingCredit * 30/100 \in \mathbf{NAT}$ **THEN**
$grantCredit := \mathbf{FALSE}$ ||
$interest := outstandingCredit * 30/100$ ||
$rr := \mathbf{FALSE}$
ELSE
$outstandingCredit := 0$ ||
$rr := \mathbf{TRUE}$
END

END

END

END

The next machine we present here is the *Customer* machine, which *sees* the *SelectProduct* and *ClientAccounts* machines described earlier. We add the invariant that *wares* is a non-empty subset of $ITEMID$. We describe below the interfaces of MACHINE *Customer*, viz:

- $aa \longleftarrow isCustomer(custID)$: Checks a valid *customerID* input to see if the number relates to a registered customer, and returns the number to indicate it does.
- $bb \longleftarrow afterSalesSupport(prod)$: Checks if product is faulty, and sets output to **TRUE** if it is. Otherwise, it does nothing.

We illustrate MACHINE *Customer* in Figure 5.24.

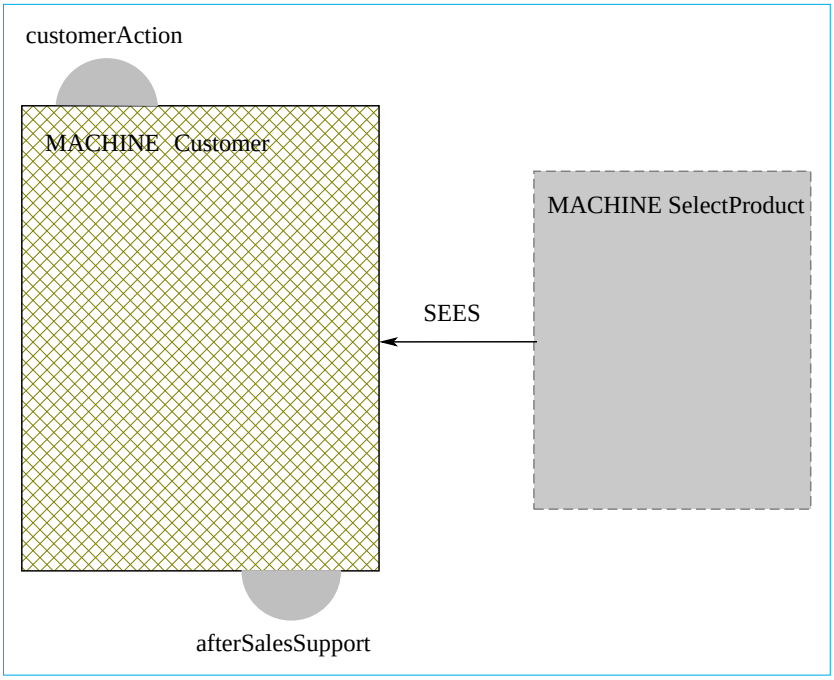


Figure 5.24: Redmound-IBM: MACHINE *Customer*

We detail the listing of MACHINE *Customer* below:

MACHINE $Customer$
SEES $SelectProduct$
DEFINITIONS
$ITEMID \triangleq 55550..99999$
VARIABLES $wares,\ returnItem,\ faulty,\ return$
INVARIANT
$wares \subseteq ITEMID \ \wedge \ returnItem \in wares \ \wedge$
$faulty \in wares \longrightarrow \mathbf{BOOL} \ \wedge \ return \in \mathbf{BOOL} \ \wedge$
$wares \neq \varnothing$
INITIALISATION
$wares,\ returnItem,\ faulty,\ return :=$
$\{55550\},\ 55550,\ \{55550 \mapsto \mathbf{FALSE}\},\ \mathbf{FALSE}$
OPERATIONS
$aa \longleftarrow isCustomer(custID) \triangleq$
PRE $custID \in NAT \ \wedge \ custID \in registered$ **THEN**
$aa := custID$
END;

$bb \longleftarrow afterSalesSupport(prod) \triangleq$
PRE $prod \in ITEMID$ **THEN**
IF $faulty(prod) = \mathbf{TRUE}$ **THEN**
$return := \mathbf{TRUE} \ \|$
$returnItem := prod \ \|$
$bb := \mathbf{TRUE}$
END
END

We next present the *PersonnelRecords*, which monitors some personnel records of employees. Since any company must have at least one employee, we add the invariant that *employeeID* is not *NULL*. We describe below the interfaces to MACHINE *PersonnelRecords*, viz:

- $zz \longleftarrow isEmployee(empID)$: Checks input data relates to a valid employee and returns *TRUE*, if so. Otherwise, the operation returns *FALSE*.

➠ $removeEmployee(empID)$: Removes details if input data relates, for example, to a former employee.

We detail the listing of MACHINE *PersonnelRecords* below:

MACHINE $PersonnelRecords$
DEFINITIONS
EMP_ID_RANGE $\triangleq$ 111100..111199
VARIABLES $employees$, $employeeName$, $employeeID$,
$employeeStatus$
INVARIANT $employees \subseteq EMP\_ID\_RANGE \wedge$
$employeeName \in$ **BOOL** $\wedge$
$employeeID \in$ EMP_ID_RANGE $\wedge$
$employeeStatus \in$ **BOOL** $\wedge$ $employeeID \notin \varnothing$
INITIALISATION $employees$, $employeeName$, $employeeID$,
$employeeStatus := \varnothing$, **FALSE**, 111100, **FALSE**
OPERATIONS
$zz \longleftarrow isEmployee(empID) \triangleq$
PRE $empID \in$ EMP_ID_RANGE **THEN**
IF $empID \in employees$ **THEN**
$employeeStatus :=$ **TRUE** ||
$zz :=$ **TRUE**
ELSE
$employeeStatus :=$ **FALSE** ||
$zz :=$ **FALSE**
END
END;

$removeEmployee(empID) \triangleq$
PRE $empID \in$ EMP_ID_RANGE **THEN**
IF $empID \notin employees$ **THEN**
$employeeStatus :=$ **FALSE** ||
$employeeName :=$ **FALSE**
END
END
END

The next machine we discuss here is the *Training_Mgt*, which monitors the attendance and training records of employees. We provide the additional invariant that *emp* is a natural number in the range 111100 to 111199, as described in the System Catalogue (Section 5.2.4.3). We describe below the interfaces to MACHINE Training_Mgt, viz:

- $ii, jj \longleftarrow getAbsenceRecords(empID,\ abs)$: Sets the absence records of employee and returns employee number and corresponding absence records.
- $kk, qq \longleftarrow getCoursesCompleted(empID,\ crsComp)$: Checks to ensure employees undertake at least one training course in the year, and returns employee number and corresponding number of courses completed. If no course has been completed to date, the operation then sets the $offerTraining$ flag to **TRUE**, indicating the need for a training course to be arranged.

We detail the listing of MACHINE *Training_Mgt* below:

MACHINE Training_Mgt(maxAttendance, maxCourses)
CONSTRAINTS
$maxAttendance = 240 \ \wedge \ maxCourses = 5$
SEES $PersonnelRecords$
DEFINITIONS
ANNUAL_ATTENDANCE == 0..240
VARIABLES
$attendanceRecords,\ absence,\ coursesCompleted,\ emp,$
$offerTraining$
INVARIANT
$attendanceRecords \in$ ANNUAL_ATTENDANCE $\wedge\ absence \in \mathbf{NAT} \wedge$
$absence \leq maxAttendance \ \wedge \ coursesCompleted \in \mathbf{NAT} \wedge$
$coursesCompleted \leq maxCourses \ \wedge emp \in \mathbf{NAT1} \wedge$
$emp \geq 111100 \ \wedge \ emp \leq 111199 \ \wedge \ offerTraining \in \mathbf{BOOL}$
INITIALISATION
$attendanceRecords,\ absence,\ coursesCompleted,\ emp,$
$offerTraining \ := \ 240,\ 0,\ 2,\ 111101,\ \mathbf{FALSE}$

OPERATIONS

$ii,\ jj \longleftarrow getAbsenceRecords(empID,\ abs) \triangleq$

PRE $empID \in \mathbf{NAT1} \ \wedge \ empID \geq 111100 \ \wedge$
$empID \leq 111199 \ \wedge \ abs \in \mathbf{NAT} \ \wedge$
$abs \leq maxAttendance$ **THEN**
$emp := empID \ \|$
$absence := abs \ \|$
$ii := empID \ \|$
$jj := abs$
END;

$kk,\ qq \longleftarrow getCoursesCompleted(empID,\ crsComp) \triangleq$

PRE $empID \in \mathbf{NAT1} \ \wedge \ empID \geq 111100 \ \wedge$
$empID \leq 111199 \ \wedge \ crsComp \in \mathbf{NAT} \ \wedge$
$crsComp \leq maxCourses$ **THEN**
IF $crsComp < 1$ **THEN**
$offerTraining := \mathbf{TRUE} \ \|$
$emp := empID \ \|$
$coursesCompleted := crsComp + 1 \ \|$
$kk := empID \ \|$
$qq := crsComp + 1$
ELSE
$emp := empID \ \|$
$coursesCompleted := crsComp \ \|$
$kk := empID \ \|$
$qq := crsComp \ \|$
END
END
END

The next machine we discuss here is the *PerfRelatedPay*, which uses the attendance records of employees to calculate their *Performance Related Pay*, i.e., the amount by which their annual salaries will be increased in the next business year. Machine *PerfRelatedPay sees* machines *PersonnelRecords* and *Training_Mgt*. We describe below the singular interface to *PerfRelatedPay*, viz:

➠ $pp \longleftarrow calculatePRP(salary,\ abs)$

We illustrate the MACHINE *PerfRelatedPay*, with its interface, in Figure 5.25.

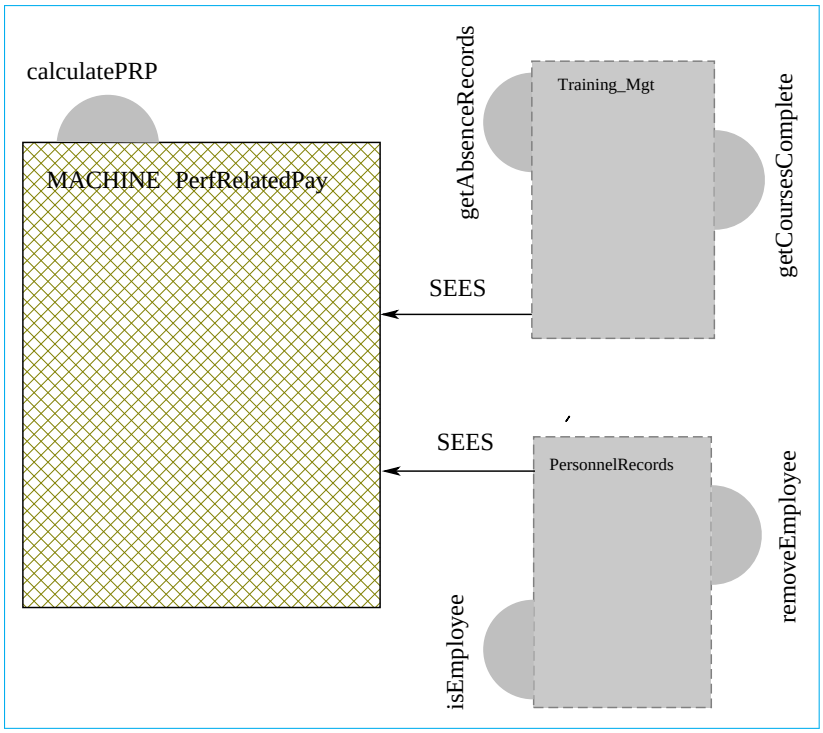


Figure 5.25: Redmound-IBM: MACHINE PerfRelatedPay

We detail the listing of MACHINE *PerfRelatedPay* below:

MACHINE $PerfRelatedPay$
SEES $PersonnelRecords$, Training_Mgt
VARIABLES
$outstanding,\ okay,\ underachieved,\ critical,$
$warning,\ discipline,\ prp$
INVARIANT
$outstanding \in \mathbf{NAT} \ \wedge \ okay \in \mathbf{NAT} \ \wedge underachieved \in \mathbf{NAT} \ \wedge$
$critical \in \mathbf{NAT} \ \wedge \ warning \in \mathbf{BOOL} \ \wedge \ discipline \in \mathbf{BOOL} \ \wedge$
$prp \in \mathbf{NAT}$
INITIALISATION

$outstanding,\ okay,\ underachieved,\ critical, warning,$
$discipline,\ prp\ :=\ 0,\ 0,\ 0,\ 0,\ \mathbf{FALSE},\ \mathbf{FALSE},\ 0$

OPERATIONS

$pp \longleftarrow calculatePRP(salary,\ abs)\ \triangleq$
PRE $salary \in \mathbf{NAT1}\ \wedge\ abs \in \mathbf{NAT}\ \wedge\ abs \leq 240\ \wedge$
$0 \leq salary * abs/100$ **THEN**
SELECT $abs \leq 3\ \wedge\ 0 \leq salary * 3/100\ \wedge$
$salary * 3/100 \leq 2147483647$ **THEN**
$prp := salary * 3/100\ \ \|$
$pp := salary * 3/100$
WHEN $abs = 4\ \vee\ abs = 5\ \wedge\ 0 \leq salary * 2/100\ \wedge$
$salary * 2/100 \leq 2147483647$ **THEN**
$prp := salary * 2/100\ \ \|$
$pp := salary * 2/100$
WHEN $abs = 6\ \vee\ abs = 7\ \wedge\ 0 \leq salary * 1/100\ \wedge$
$salary * 1/100 \leq 2147483647$ **THEN**
$prp := salary * 1/100\ \ \|$
$pp := salary * 1/100\ \ \|$
$warning := \mathbf{TRUE}$
ELSE
$prp := 0$
$pp := 0$
$discipline := \mathbf{TRUE}$
END
END

END

We now discuss the *StaffSalaries* machine, which has just the one interface used to monitor staff loans and post-tax deductions. This interface, $ee, gg \longleftarrow\ requestLoan(amount)$ accepts a loan request input parameter, and checks defined business rules to determine if loan is to be granted or rejected. If loan is granted, then repayment amount and salary after deduction of *monthly* repayment amount is calculated.

To the **INVARIANT** clause of MACHINE *StaffSalaries* we added the requirement that employees may only be granted loan up to 25% (i.e., a quar-

ter) of their salaries. We illustrate the MACHINE *StaffSalaries*, with its interface, in Figure 5.26.

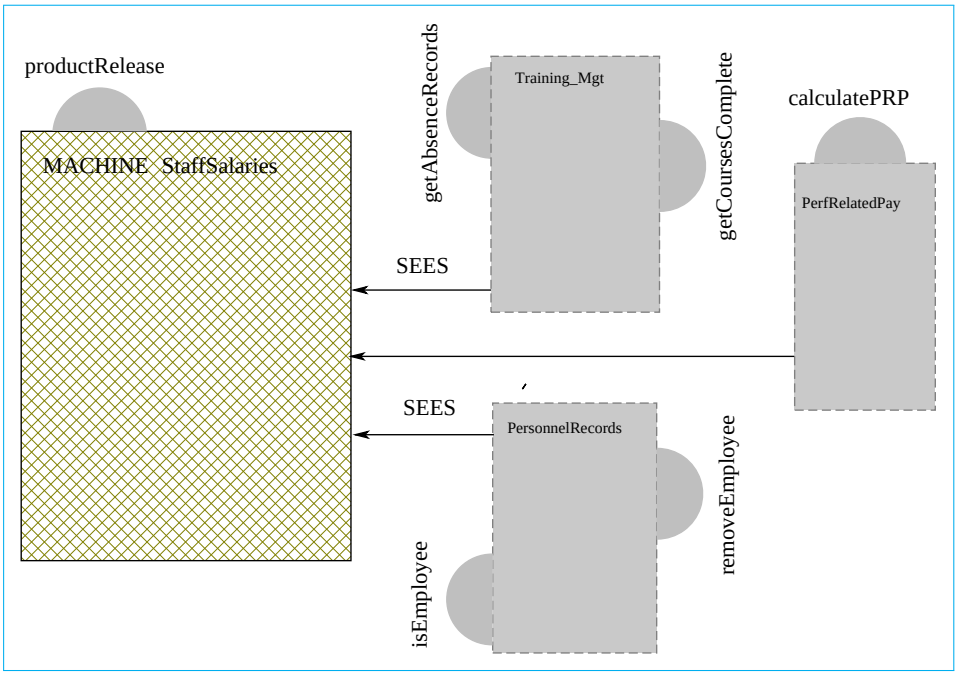


Figure 5.26: Redmound-IBM: MACHINE StaffSalaries

Note that the output variable *gg* is intended to indicate whether the employee in question is credit-worthy or not. We detail the listing of MACHINE *StaffSalaries* below:

MACHINE $StaffSalaries$
SEES $Training\_Mgt,\ PersonnelRecords,\ PerfRelatedPay$
SETS
$LOANSTATUS\ =\ \{Owing,\ NotOwing\}$
PROPERTIES
$LOANSTATUS\ \neq\ \varnothing$
DEFINITIONS
$EMPID\ \triangleq$ 111100..111199
VARIABLES
$identifier,\ loanStatus,\ outstandingLoan,\ requestedLoan,$

$repaymentAmount,\ salaryB4Deduction,\ salaryAfter,$
$grantLoan$

INVARIANT

$identifier \subseteq EMPID \wedge$
$loanStatus \in identifier \longrightarrow LOANSTATUS \wedge$
$outstandingLoan \in \mathbf{NAT} \ \wedge \ requestedLoan \in \mathbf{NAT} \wedge$
$repaymentAmount \in \mathbf{NAT} \ \wedge \ salaryB4Deduction \in \mathbf{NAT} \wedge$
$salaryAfter \in \mathbf{NAT} \ \wedge \ grantLoan \in \mathbf{BOOL} \wedge$
$outstandingLoan \leq salaryB4Deduction/4$

INITIALISATION

$identifier,\ loanStatus,\ outstandingLoan,\ requestedLoan,$
$repaymentAmount,\ salaryB4Deduction,\ salaryAfter,$
$grantLoan \ :=$
$\{111100\},\ \{111100 \mapsto NotOwing\},\ 0,\ 0,\ 0,\ 0,\ 0,\ \mathbf{FALSE}$

OPERATIONS

$ee, gg \longleftarrow \ requestLoan(empID) \ \triangleq$
PRE $empID \in \mathbf{NAT1} \ \wedge \ empID \in identifier$ **THEN**
ANY $amount$ **WHERE** $amount \in \mathbf{NAT} \wedge$
$outstandingLoan + amount \leq 2147483647 \wedge$
$salaryB4Deduction/4 \ \leq \ 2147483647 \wedge$
$0 \ \leq \ outstandingLoan/6 \wedge$
$0 \ \leq \ salaryB4Deduction -$
$repaymentAmount$ **THEN**
IF $outstandingLoan = 0 \ \wedge \ amount = 0$ **THEN**
$loanStatus(empID) := NotOwing \ \|$
$ee := empID \ \|$
$gg := \mathbf{TRUE}$
ELSIF $outstandingLoan \ > \ 0 \wedge$
$outstandingLoan{+}amount \leq salaryB4Deduction/4$
THEN
$loanStatus(empID) := Owing \ \|$
$grantLoan \ := \ \mathbf{TRUE} \ \|$
$outstandingLoan \ := \ outstandingLoan{+}amount$
$\| \ repaymentAmount \ := \ outstandingLoan/6 \ \|$
$salaryAfter \ := \ salaryB4Deduction -$
$repaymentAmount \ \|$

$ee := empID \parallel$
$gg := \textbf{TRUE}$
ELSE
$loanStatus(empID) := Owing \parallel$
$grantLoan := \textbf{FALSE} \parallel$
$salaryAfter := salaryB4Deduction - repaymentAmount \parallel$
$ee := empID \parallel$
$gg := \textbf{FALSE}$
END
END
END
END

The next sub-component we discuss here is the *Loan_CredAccounts* machine, which sees the *StaffSalaries* and *ClientAccounts* machines. We describe below the interfaces to *Loan_CredAccounts*, viz:

- $ll \longleftarrow checkLoanAccount(empID)$: This operation checks *StaffSalaries* machine to determine if employee has any outstanding loan, and outputs the loan status, i.e., whether the employee is owing the company or not.
- $mm \longleftarrow checkCreditAccount(custID)$: The operation checks *ClientAccounts* machine to determine if the customer has any outstanding credit, and outputs the credit status, i.e., whether the customer is owing the company or not.

We illustrate the MACHINE *Loan_CredAccounts*, with its interfaces, in Figure 5.27.

We detail the listing of MACHINE *Loan_CredAccounts* below:

MACHINE *Loan_CredAccounts*
SEES *StaffSalaries*, *ClientAccounts*
VARIABLES *loanSts*, *creditStatus*
INVARIANT

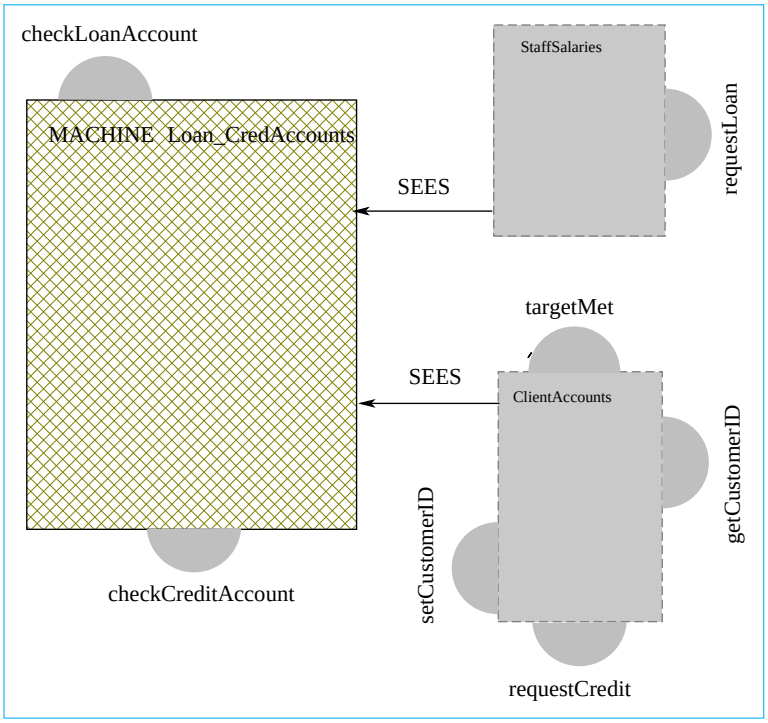


Figure 5.27: Redmound-IBM: MACHINE Loan_CredAccounts

$loanSts \in LOANSTATUS \wedge creditStatus \in \mathbf{NAT}$

INITIALISATION

$loanSts,\ creditStatus := Owing,\ 1$

OPERATIONS

$ll \longleftarrow checkLoanAmount(empID) \triangleq$

PRE $empID \in \mathbf{NAT1} \wedge empID \in identifier$ **THEN**

IF $loanStatus(empID) \in LOANSTATUS$ **THEN**

$loanSts := loanStatus(empID) \parallel$

$ll := loanStatus(empID)$

END

END;

$mm \longleftarrow checkCreditAccount(custID) \triangleq$

PRE $custID \in \mathbf{NAT1} \wedge custID \in registered$ **THEN**

$creditStatus := outstandingCredit \parallel$

$mm := outstandingCredit$

END

END

We next present the *Accounts* machine, which extends the *StaffSalaries*, *ClientAccounts* and *Loan_CredAccounts* machines. We describe below the interfaces to the *Accounts* machine, viz:

- $xx, yy \longleftarrow monitorLoanAccounts(empID)$: The operation checks *StaffSalaries* machine and calls the $xx, yy \longleftarrow requestLoan(empID)$ operation.
- $dd \longleftarrow monitorClientAccounts(custID)$: The operation checks *ClientAccounts* machine to determine if the customer is registered or not. If customer is registered control is passed to the $dd \longleftarrow checkCreditAccount(custID)$ of machine *Loan_CredAccounts*. If customer is not registered, then a temporary guestAccount is opened for the customer.

We illustrate the MACHINE *Accounts*, with its interfaces and included machines, in Figure 5.28.

The listing of the *Accounts* machine is detailed below:

MACHINE *Accounts*
INCLUDES
$StaffSalaries$, $ClientAccounts$, $Loan\_CredAccounts$
SETS
$GUESTACCOUNTS = \{ga11001,\ ga11002,\ ga11003,\ ga11004,\ ga11005\}$
PROPERTIES
$GUESTACCOUNTS \neq \varnothing$
VARIABLES
$guestAccount$
INVARIANT
$guestAccount \in GUESTACCOUNTS$
INITIALISATION
$guestAccount := ga11001$

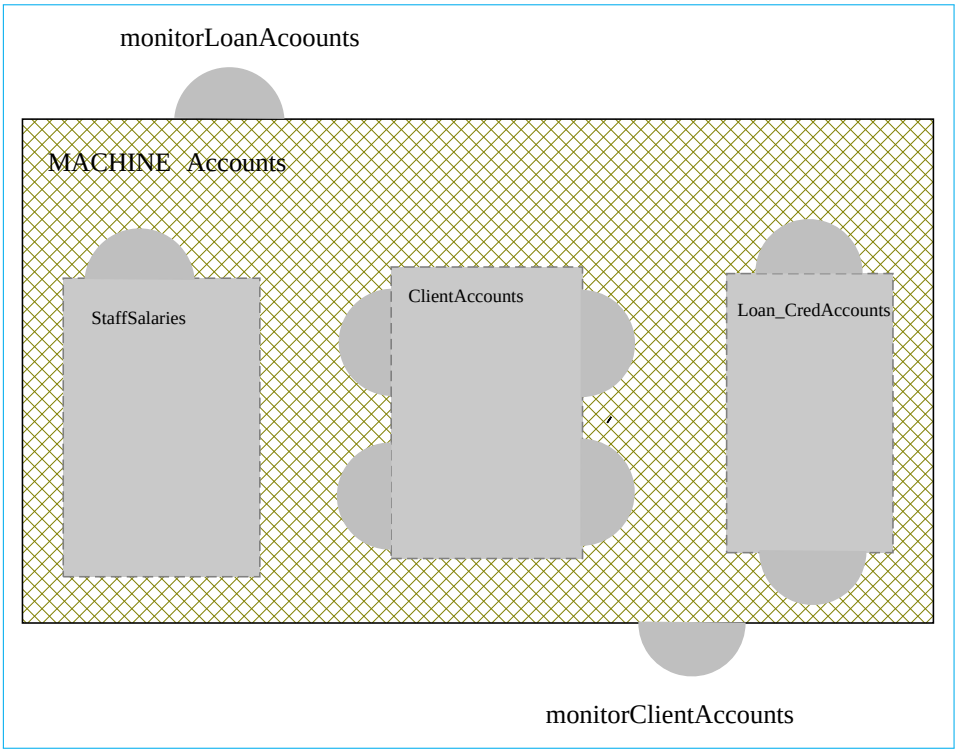


Figure 5.28: Redmound-IBM: MACHINE Accounts

OPERATIONS

$dd \longleftarrow monitorClientAccounts(custID) \triangleq$

PRE $custID \in$ **NAT1 THEN**

SELECT $custID \in registered$ **THEN**

$dd \longleftarrow checkCreditAccount(custID)$

WHEN $custID \notin registered$ **THEN**

ANY $guestID \in GUESTACCOUNTS$ **THEN**

$guestAccount := guestID \ \|$

$dd := 0$

END

ELSE

$dd := 1$

END

END;

$xx, yy \longleftarrow monitorLoanAccounts(empID) \triangleq$

PRE $empID \in$ **NAT1** $\wedge$ $empID \in identifier$ **THEN**

$xx, yy \longleftarrow requestLoan(empID)$

END

END

The next machine we discuss here is the *Ts_ProductRelease* machine, which provides the singular interface, $pr \longleftarrow productRelease(pID)$, for monitoring the business rules for releasing purchased products to customers.

We illustrate the MACHINE *Ts_ProductRelease*, with its interface and seen machines, in Figure 5.29.

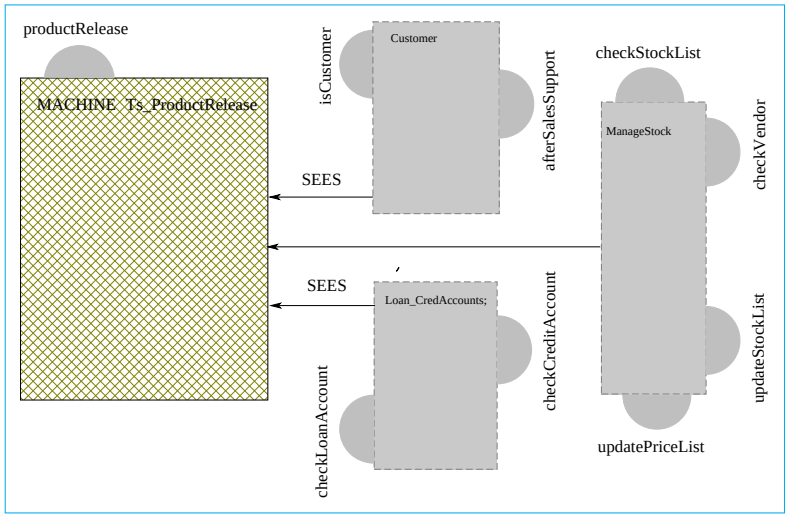


Figure 5.29: Redmound-IBM: MACHINE Ts_ProductRelease

We define within the **INVARIANT** clause of *Ts_ProductRelease* a number of additional requirements, for example, that a customer must not be charged shipping cost whenever he has opted to collect product from the store (i.e., *toShip* = **FALSE**), and that whenever the weight of the product is over 10kg, the shipping cost must be over £10 (subject to special delivery arrangements to be made with courier). The listing of the *Ts_ProductRelease* machine is detailed below:

MACHINE $Ts\_ProductRelease$

SEES $Customer$, $Loan\_CredAccounts$, $ManageStock$

VARIABLES

$paymentConfirmed$, $toShip$, $weight$, $shippingCost$, $documentsAdded$

INVARIANT

$paymentConfirmed \in \mathbf{BOOL} \ \wedge \ toShip \in \mathbf{BOOL} \ \wedge$
$weight \in \mathbf{NAT1} \ \wedge \ weight \leq 10 \ \wedge$
$shippingCost \in \mathbf{NAT} \ \wedge \ documentsAdded \in \mathbf{BOOL} \ \wedge$
$!ts \cdot (ts = toShip \wedge ts = \mathbf{FALSE} \ \Rightarrow \ shippingCost = 0) \ \wedge$
$!ww \cdot (ww = weight \wedge ww > 10 \ \Rightarrow \ shippingCost > 0) \ \wedge$
$\mathbf{not}(paymentConfirmed = \mathbf{TRUE} \wedge toShip = \mathbf{FALSE}) \ \wedge$
$\mathbf{not}(paymentConfirmed = \mathbf{TRUE} \wedge toShip = \mathbf{TRUE}) \ \wedge$
$weight \leq 10$

INITIALISATION

$paymentConfirmed$, $toShip$, $weight$, $shippingCost$, $documentsAdded$:= **FALSE**, **FALSE**, 1, 0, **FALSE**

OPERATIONS

$pr \longleftarrow productRelease(pID) \ \triangleq$
PRE $pID \in \mathbf{NAT1} \ \wedge \ pID \in stockList \ \wedge$
$\mathbf{not}(paymentConfirmed = \mathbf{FALSE} \wedge toShip = \mathbf{TRUE})$ **THEN**
IF $paymentConfirmed = \mathbf{TRUE} \wedge toShip = \mathbf{FALSE}$ **THEN**
$shippingCost := 0 \ \|$
$documentsAdded := \mathbf{TRUE} \ \|$
$pr := \mathbf{TRUE}$
ELSIF
$paymentConfirmed = \mathbf{TRUE} \wedge toShip = \mathbf{TRUE}$ **THEN**
SELECT $weight > 0 \wedge weight < 2$ **THEN**
$shippingCost := 5 \ \|$
$documentsAdded := \mathbf{TRUE} \ \|$
$pr := \mathbf{TRUE}$
WHEN $weight \geq 2 \wedge weight < 5$ **THEN**

$shippingCost := 8 \ \|$
$documentsAdded := \textbf{TRUE} \ \|$
$pr := \textbf{TRUE}$
WHEN $weight \geq 5 \wedge weight \leq 10$ **THEN**
$shippingCost := 10 \ \|$
$documentsAdded := \textbf{TRUE} \ \|$
$pr := \textbf{TRUE}$
ELSE
$shippingCost := 10000 \ \|$
$pr := \textbf{FALSE}$
ELSE
$shippingCost := 10000 \ \|$
$pr := \textbf{FALSE}$
END
END
END

Next, we discuss here the *Sales_Technical* machine, which includes the *ManageStock* and *Ts_ProductRelease* machines. Further, the *Sales_Technical* machine promotes the operations *checkStockList, checkVendor,* and *productRelease* to full-fledged operations of the including machine. We describe below the interfaces to the *Sales_Technical* machine, viz:

- $vv \longleftarrow monitorLocalStock(product, option)$: The operation checks *ManageStock* machine and calls the *checkLocalStock* operation.
- $ww \longleftarrow monitorVendorStock(product, option)$: The operation checks *ManageStock* machine and passes control to the $checkVendor$ operation.
- $uu \longleftarrow monitorProductRelease(pID)$: This operation checks the *Ts_ProductRelease* machine, and passes control to the *productRelease* operation of the *Ts_ProductRelease* machine.

We illustrate the MACHINE *Sales_Technical*, with its interfaces, in Figure 5.30.

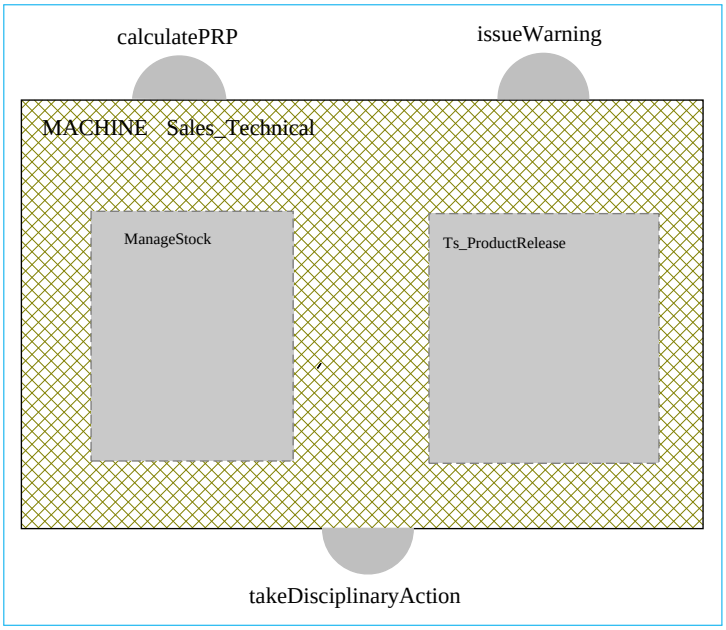


Figure 5.30: Redmound-IBM: MACHINE Sales_Technical

The listing of the *Sales_Technical* machine is detailed below:

MACHINE $Sales\_Technical$
INCLUDES $ManageStock,\ Ts\_ProductRelease$
PROMOTES
$checkStockList,\ checkVendor,\ productRelease$
VARIABLES $stockState$
INVARIANT $stockState \in \mathbf{BOOL}$
INITIALISATION $stockState := \mathbf{BOOL}$
OPERATIONS
$vv \longleftarrow monitorLocalStock(product,\ option)\ \triangleq$
PRE $product \in \mathbf{NAT1}\ \wedge\ product \in stockList\ \wedge$
$option \in OPTION\_LIST\ \wedge\ vv \in \mathbf{BOOL}$ **THEN**
$vv \longleftarrow checkStockList(product,\ option)\ \|$
$stockState := vv$
END;

$ww \longleftarrow monitorVendorStock(product,\ option)\ \triangleq$

PRE $product \in \mathbf{NAT1} \ \wedge \ product \in stockList \ \wedge$
$option \in OPTION\_LIST$ **THEN**
$ww \longleftarrow checkVendor(product, \ option)$
END;

$uu \longleftarrow monitorProductRelease(pID) \ \triangleq$
PRE $pID \in \mathbf{NAT1} \ \wedge \ pID \in stockList \ \wedge$
not$(paymentConfirmed = \mathbf{FALSE} \ \wedge \ toShip = \mathbf{TRUE})$
THEN
$uu \longleftarrow productRelease(pID)$
END

END

The next machine component we present here is the *Overheads* machine, which monitors the petty cash imprest account and updates it according to the defined business rules every 30 days.

The detailed listing of the *Overheads* machine is given below:

MACHINE *Overheads*
CONSTANTS *limit*
PROPERTIES $limit = 30$
DEFINITIONS $PETTYCASH == 100$
VARIABLES *monthCounter*, *balance*, *pettyCash*
INVARIANT
$monthCounter \leq limit \ \wedge \ balance \in \mathbf{NAT} \ \wedge$
$pettyCash \in \mathbf{NAT} \ \wedge \ pettyCash \leq PETTYCASH$
INITIALISATION
$monthCounter, \ balance, \ pettyCash \ := 1, \ 0, \ 0$
OPERATIONS
$setPettyCash(counter) \ \triangleq$
PRE $counter \in \mathbf{NAT1} \ \wedge \ counter = limit$ **THEN**
IF $pettyCash < PETTYCASH$ **THEN**
$balance := pettyCash \ \|$
$pettyCash := PETTYCASH$

END

END

END

The last sub-component discussed here is the *Administration* machine, which includes the *Overheads*, *Training_Mgt* and *PersonnelRecords* machines. Further, the *Administration* machine is parameterised with the same parameters that the *Training_Mgt* machine receives. We describe below the interfaces to the *Administration* machine, viz:

- $monitorPettyCash(counter)$: The operation checks the *Overheads* machine and calls the $setPettyCash$ operation.
- $ff, hh \longleftarrow monitorAbsenceRecords$: The operation checks *Training_Mgt* machine and passes control to the $getAbsenceRecords(empID, abs)$ operation.
- $nn, mz \longleftarrow monitorCoursesCompleted(empID, crsComp)$: This operation checks the *getCoursesCompleted(empID, crsComp)* operation of the *Training_Mgt* machine.

We illustrate the MACHINE *Administration*, with its interfaces, in Figure 5.31.

The listing of the *Administration* machine is detailed below:

MACHINE $Administration(maxAttendance,\ maxCourses)$
INCLUDES
$Overheads,\ Training\_Mgt(maxAttendance,\ maxCourses),$
$PersonnelRecords$
CONSTRAINTS
$maxAttendance = 240\ \wedge\ maxCourses = 5$
OPERATIONS
$monitorPettyCash(counter)\ \triangleq$
PRE $counter \in$ **NAT1** $\wedge\ counter = limit$ **THEN**
$setPettyCash(counter)$
END;

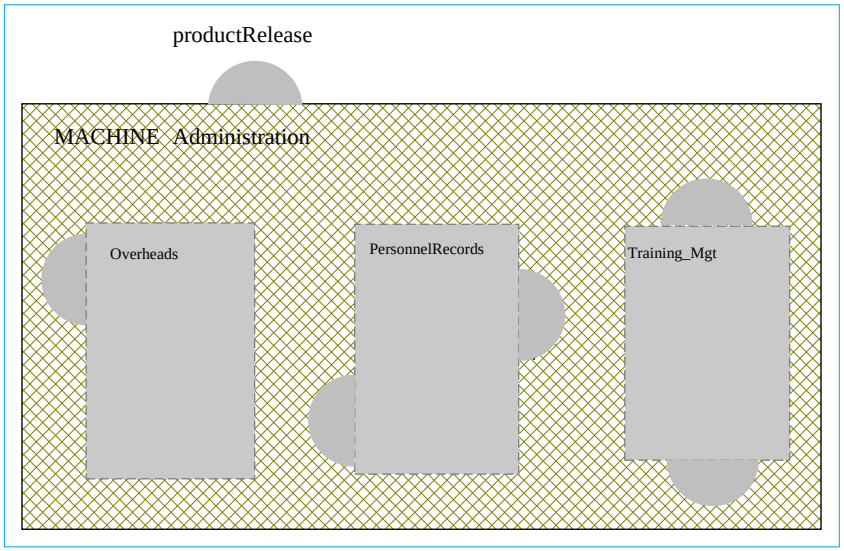


Figure 5.31: Redmound-IBM: MACHINE Administration

$$ff, hh \longleftarrow monitorAbsenceRecords(empID,\ abs) \ \triangleq$$
$$\textbf{PRE}\ empID \in \textbf{NAT1}\ \wedge\ empID \in employees\ \wedge$$
$$abs \in \textbf{NAT}\ \wedge\ abs \leq maxAttendance\ \textbf{THEN}$$
$$ff, hh \longleftarrow getAbsenceRecords(empID,\ abs)$$
$$\textbf{END};$$

$$nn, mz\ \longleftarrow monitorCoursesCompleted(empID,\ crsComp)\ \triangleq$$
$$\textbf{PRE}\ empID \in \textbf{NAT1}\ \wedge\ empID \in employees\ \wedge$$
$$crsComp \in \textbf{NAT}\ \wedge\ crsComp \leq maxCourses\ \textbf{THEN}$$
$$nn, mz\ \longleftarrow getCoursesCompleted(empID,\ crsComp)$$
$$\textbf{END}$$
$$\textbf{END}$$

Having presented the formal B Machine specification of the subsystems of Redmound-IBM, we discuss in Section 5.4.2 below the results of analysing the flow of information between machine variables, using our information flow analyser, with respect to the security flow lattice defined in Section 5.3.

5.4.2 Redmound-IBM Machines: Information Flow Analysis

In this section, we run the formal B machines defined in Section 5.4.1 through our information flow analyser, BMethalizer.exe, developed using C++. To make the analysis even more interesting, we assume that some of the variables, for example, yy, $guestID$, nn, aa, ... are not classified in the security flow lattice. This will enable us to check if flows non-public[4] variables to such variables are regarded '*insecure*', as they should. We will also review our analysis results to see if flows to/from *incomparable* variables are correctly captured by the flow analyser. Our Information Flow Analyser takes two input files. The first file contains the ascii version of the B Machines described earlier in Section 5.4.1. The second file contains the text version of the security flow policy defined in Section 5.3. This security flow policy we define on a single flow lattice, as illustrated in Figure 5.20. For simplicity in this text version of the security flow policy, we write 0 to denote the $\varnothing$ security class; A for the $[A]$ security class; B for $[B]$; C for $[C]$; AB for $[AB]$; AC for $[A]$; and ABC for $[ABC]$.

Now suppose we have two variables $var1$ and $var2$ such that $var2$ depends on $var1$, i.e., information flows from $var1$ into $var2$. The output of our information flow analyser writes $var2 \mapsto var1$ to denote that $var2$ depends on $var1$. Any flow adjudged *insecure* by our information flow analyser is highlighted in bold red, to enhance visibility of parts of the system (or security flow policy) that may need to be modified. Note that all screenshot figures referenced in this section are presented in Appendix D. The first machine we analyse is '*Accounts*'.

The screenshot showing the output on passing the ascii version of the B machine *Accounts* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 1 in Appendix D. Figure 2 shows some of the flows between variables in the $dd \longleftarrow monitorClientAccounts(custID)$ operation of Accounts. Notice that the flow from $guestID$ into $guestAccount$ is adjudged *insecure*. This is the case because we intentionally leave the variable $guestID$ undefined in Section 5.3, hence the two variables are incomparable. Similarly, since yy is also undefined in Section 5.3, all flows between yy and

[4]Variables with security classification higher than $\varnothing$.

other non-public variables in the $xx, yy \longleftarrow monitorLoanAccounts(empID)$ operation are incomparable and hence *insecure*, as shown by the screenshot in Figure 3 of Appendix D. The case of flows within Accounts is also very informative as they entail flows due to B structuring mechanisms via the **EXTENDS** clause, which made it possible for operations of extended machines to expose variables of the visible machines as read-only to operations of the machine *Accounts*. The **EXTENDS** clause also made variables (e.g., $registered$ and $identifier$) of the visible machines directly visible as read-only within operations of the machine *Accounts*.

On passing the ascii version of the B machine *Administration* and the security policy file *secPol_RIBM.txt* to the flow analyser, we get the screenshot displayed in Figure 4 of Appendix D. Figures 5 and 6 are relatively straightforward as they show that all the flows concerned are *secure*. However, as depicted in Figure 7 of Appendix D, the flow from $crsComp$ into nn is adjudged *insecure* as we expect since $crsComp$ has the security classification $[AC]$, whereas the security classification of nn is not defined in Section 5.3.

Figure 8 shows the screenshot on passing the ASCII version of the B machine *ClientAccounts* and the security flow policy file *secPol_RIBM.txt* to our information flow analyser. Notice that the screenshots shown in Figures 10 and 11 are straightforward since all flows displayed therein are secure. Notice, however, that the screenshot showing the analysis of operation $cc \longleftarrow getCustomerID(cId)$ of *ClientAccounts* indicates all flows into the output variable cc are insecure. This is expected because cc is a public variable with the least security classification, $\varnothing$, whereas $registered$, cId, and $CUSTOMERID$ all have a higher security classification, $[B]$. Thus for secure flows, the operation may be redesigned so that none of the variables with a higher security classification are used in the **PRE** condition or the **IF** condition (to prevent *implicit flows* into cc). Alternatively, the variable cc may be reclassified to a security level *at least as high as* $[B]$.

On passing the ascii version of the B machine *Customer* and the security policy file *secPol_RIBM.txt* to the flow analyser, we get the screenshot displayed in Figure 12 of Appendix D. The screenshot showing the analysis of operation $aa \longleftarrow isCustomer(custID)$ of the *Customer* machine, Figure

13, shows that implicit flows from *custID* and *registered* into the output variable *aa* are insecure. This is because whereas *custID* and *registered* both have a security classification of $[B]$, the variable *aa* is *unclassified* in our security flow lattice in Section 5.3. A close examination of Figure 14 of Appendix D, our security flow analysis output with respect to operation $bb \longleftarrow afterSalesSupport(prod)$ of the *Customer* machine, reveals:

i. The flow from $ITEMID$ into *bb* is insecure because $ITEMID$ is undefined;

ii. The flow from $faulty(prod)$ into *bb* is insecure because $faulty$ has a security classification of $[B]$, whereas *bb* has a security classification of $[A]$, hence are incomparable;

iii. The flow from *prod* into *bb* is secure because *prod* has classification $\varnothing$ whereas *bb* has security classification $[A]$, which is higher than $\varnothing$;

iv. For the same reason as 1 above, flows from $ITEMID$ into $return$ and $returnItem$ are insecure;

v. It is easy to see why other flows in the screenshot are adjudged secure because they are either flows between variables with the same security classification, or flows from variables with lower security classification.

The screenshot showing the output on passing the ascii version of the B machine *Loan_CredAccounts* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 15 in Appendix D. Figures 16 and 17 show that all flows within both operations of the machine are secure. This is expected, because a close examination of the machine and the defined security policy shows that all flows into updated variables of the machine are from variables with security classification lower than or equal to that of the variables being updated.

When we passed the ascii version of the B machine *ManageStock* and the security policy file *secPol_RIBM.txt* to our flow analyser, the output is as shown in Figure 18 of Appendix D. Figure 19 captures the fact that the flow from $VALID\_SID$ into *bb* is insecure, as expected, because $VALID\_SID$ is undefined in our security policy file whereas *bb* has a security classification

of $[A]$, hence both variables are incomparable. The same is true of the flow from $VALID\_SID$ into *cc* (and the flow from $VALID\_SID$ into the variable *discontinued*) in Figure 20. However, because *product* has a security classification of $[A]$ whereas *cc* has a *lower* security classification of $\varnothing$, the flow from *product* into *cc* is correctly adjudged insecure as shown in Figure 20. For the same reason, the flow from *product* into *discontinued* is correctly adjudged insecure too. Figure 21 yields some interesting results, namely:

i. Because both *priceList* and $VALID\_SID$ are undefined in our security policy file, our analyser discounts the analysis of information flow between them and other similarly unclassified variables, yielding the default assumption in such cases that the flow is secure. We *Do Not Care* about flows between these variables.

ii. Because *prod* is defined as a *public* variable (i.e., $\varnothing$ classification), any flow from *prod* is always secure. Hence the flow into the unclassified variable *stockList* is correctly adjudged secure. The intuition behind this is that it is acceptable for a *public* variable to be stored in any variable of the user's choice - even into an unclassified variable.

iii. Because both *prc* and $PRICE\_RANGE$ are mapped to the *non-public* security class $[A]$, and *priceList* is unclassified, flows from the former variables into the latter are correctly adjudged insecure.

On passing the ascii version of the B machine *Overheads* and the security policy file *secPol_RIBM.txt* to the flow analyser, we get the screenshot displayed in Figure 22 of Appendix D. And Figure 23 shows all flows within this machine are secure, hence there is nothing very interesting to discuss further.

The screenshot showing the output on passing the ascii version of the B machine *PerfRelatedPay* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 24 in Appendix D. Figure 25 of Appendix D shows that flows from *salary* into each of the variables *discipline*, *pp*, *prp*, and *warning* are insecure. This is because *salary* has a higher security classification, $[ABC]$, than these other variables, which have a security classification of $[AC]$. On the other hand, flows from *abs* into each of the same variables are correctly adjudged secure because *abs* has a lower comparable

security classification, $[C]$.

We next discuss the screenshot showing the output on passing the ascii version of the B machine *PersonnelRecords* and the security policy file *secPol_RIBM.txt* to the flow analyser, presented in Figure 26 in Appendix D. Figure 27 shows that our flow analyser adjudged flows between EMP_ID_RANGE and *zz*; from *empID* into *zz*; and from *employees* into *zz* as insecure. This is the expected result since EMP_ID_RANGE, *empID*, and *zz* all have security classification of $[C]$ whereas *zz* has a security classification of $[A]$, hence they are incomparable. The screenshot depicted in Figure 28 needs no further discussion; all flows there are adjudged secure.

On passing the ascii version of the B machine *Sales_Technical* and the security policy file *secPol_RIBM.txt* to the flow analyser, we get the screenshot displayed in Figure 29 of Appendix D. As noted earlier, for the reason that $VALID\_SID$ is undefined in our security policy file, the flows from $VALID\_SID$ into *stockState* and *vv* respectively are correctly adjudged insecure by our flow analyser as shown in Figure 30. The same is true of the flows from $VALID\_SID$ into *discontinued* and *ww* respectively within the operation $ww \longleftarrow monitorVendorStock(product, option)$ of *Sales_Technical* as depicted in Figure 31. In addition since *product* has a security classification of $[A]$, whereas *discontinued* has a classification of $\varnothing$, our flow analyser correctly adjudge the flow from *product* into *discontinued* as insecure. Also since pID[5] is undefined in our security flow policy file, the flow from pID into *uu* in the operation $uu \longleftarrow monitorProductRelease(pID)$ of *Sales_Technical* is adjudged insecure as shown in Figure 32.

The screenshot showing the output on passing the ascii version of the B machine *SelectProduct* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 33 in Appendix D. Figure 34 shows that all the flows between variables in operation $po \longleftarrow selectProduct(prod)$ are secure. In operation $op \longleftarrow selectOptions(option)$, flows from $OPTIONS$ to *op* and *selectedOptions* respectively are judged insecure, as shown in Fig-

[5]Note: because we designed our flow analyser to do case-sensitive tokenisation of variable names in B GSL, the variable *pId*, which is defined in our security flow policy file will be seen as different from pID, which is undefined.

ure 35, for the reason that *op* and *optionSelected* are both *public* variables, whereas $OPTIONS$ has a security classification of $[B]$ in our security flow policy file. For the same reason, flows from $OPTIONS$ into $productID$ and *ss* respectively are shown to be insecure in Figure 36.

The screenshot showing the output on passing the ascii version of the B machine *StaffSalaries* and the security policy file *secPol_RIBM.txt* to the flow analyser, presented in Figure 37 in Appendix D, is the next subject of our discussion. In Figure 38, we see that our analyser adjudged the flow from *amount* (security class $[B]$) into *ee* (security class $[C]$) as insecure because both variables are incomparable. The same is true of the flow from *amount* into *gg*. However, the flows from *outstandingLoan*, *repaymentAmount*, and *salaryB4Deduction* into *ee* are adjudged insecure because the former variables have higher security classification, $[ABC]$, than the latter. The screenshot shown in Figure 39 is straightforward, as all flows therein are correctly adjudged secure.

On passing the ascii version of the B machine *Training_Mgt* and the security policy file *secPol_RIBM.txt* to the flow analyser, we get the screenshot displayed in Figure 40 of Appendix D. Figures 41 and 43 are straightforward, because all flows therein are correctly adjudged secure, however, Figure 42 shows the flow from $crsComp$ into *emp* in operation
$kk, qq \longleftarrow getCoursesCompleted(empID, crsComp)$ is insecure because
$crsComp$ has a higher security classification, $[AC]$, than *emp*, which has a security classification $[C]$.

The screenshot showing the output on passing the ascii version of the B machine *Ts_ProductRelease* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 44 in Appendix D. For the reason that pID (as discussed earlier) is undefined in our security policy file, Figure 45 shows the flow from pID into *shippingCost* to be correctly adjudged insecure.

And finally, the screenshot showing the output on passing the ascii version of the B machine *Vendors* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 46 in Appendix D. In our security policy

file, *stocked* has a security classification $[A]$ whereas $PRICE$ is undefined. Hence Figure 47 shows our flow analyser adjudged the flow from $PRICE$ into *stocked* in operation *setProduct* to be insecure. Similarly, the flow from $PRODUCTS$ into *bb* is adjudged insecure as shown in Figure 48 because $PRODUCTS$ is unclassified, whereas *bb* has a security classification of $[A]$.

From our discussion of the analysis of information flow within the B machines used in the development of Redmound-IBM, we have seen that our information flow analysis framework is able to successfully identify:

i. Insecure *explicit* flows due to direct substitutions;

ii. Insecure *implicit* flows through dependency on variables used in conditional predicates of **IF**, **PRE**, etc. substitutions;

iii. Insecure flows due to variable visibility through structuring mechanisms in B; and

iv. Insecure flows between incomparable variables, with respect to defined security policy.

In Section 5.5, we will explore information flow between variables in the B Refinements of some of the machines discussed in section 5.4.1.

5.5 Redmound-IBM: The Refinements

We present in this section the B Refinements of some of the machines defined in Subsection 5.4.1. The refinements (i.e., $ManageStock\_r$ and $StaffSalaries\_r$ respectively) of the machines $ManageStock$ and $StaffSalaries$ have been selected because they are arguably among the least trivial choices we could make. To satisfy the refinement relation, both refinements have the same number of interfaces and functionality as the corresponding machines being refined (i.e., $ManageStock$ and $StaffSalaries$ respectively). Also, because the variables in the refinements are linked with the variables in the corresponding machines (via linking invariants), we assume the same security classification for the linked variables. The B Refinement listings are presented in Subsection 5.5.1, whereas the analysis results using our information flow analyser are discussed in Subsection 5.5.2.

5.5.1 Redmound-IBM: Formal B Refinements

A screenshot of the Redmound-IBM project in Atelier B showing the proof obligations automatically generated and proved (with the added refinements highlighted) is given in Figure 5.32[6].

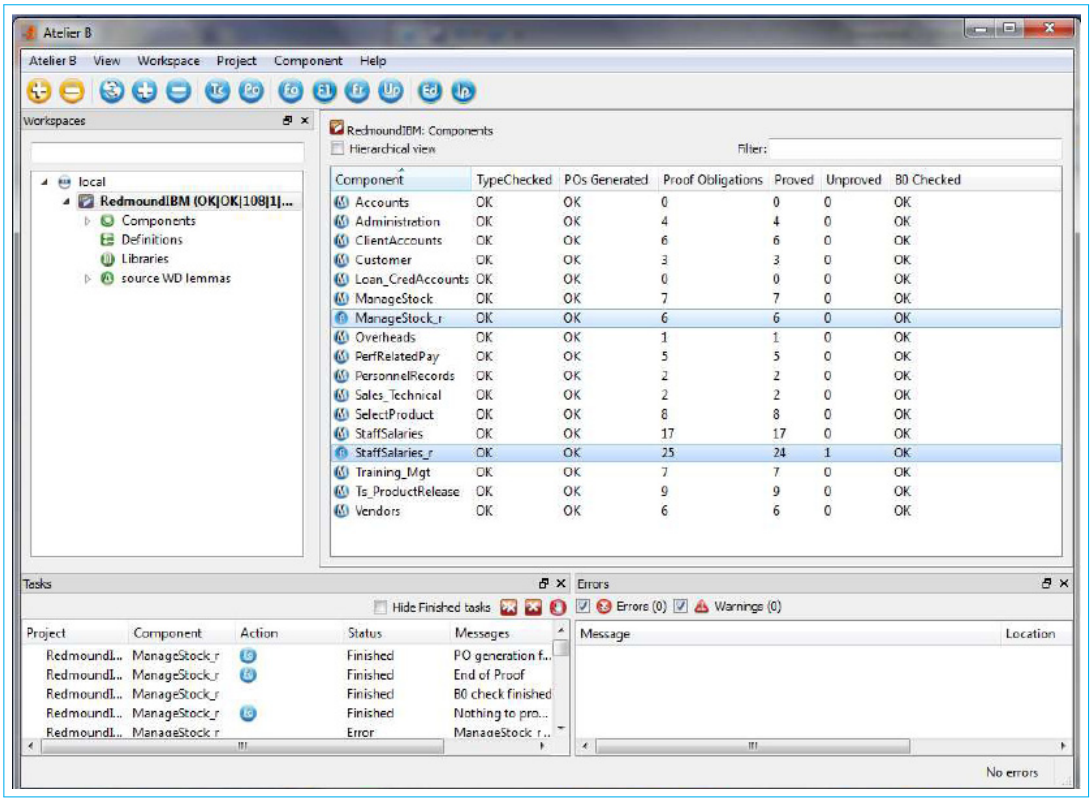


Figure 5.32: Redmound-IBM: Proof Obligations Consistency Check

We begin our discussion here with a listing of *ManageStock_r* refinement of the *ManageStock* machine.

REFINEMENT *ManageStock_r*
REFINES

[6]We encountered some difficulty proving one of the twenty-five proof obligations generated for the refinement *StaffSalaries_r*, as you can see from the figure. However, this does not adversely impact the primary objective of the case-study, which is (in this instance) the analysis of information flow within B Refinements.

$ManageStock$

SEES

$Vendors,\ SelectProduct$

VARIABLES $stockID\_r,\ stockList\_r,$

$discontinued\_r,\ price\_r,\ priceList\_r$

DEFINITIONS

$VALID\_SID \triangleq 55550..99999;$

$PRODUCT\_LIST \triangleq 55550..99999;$

$PRICE\_RANGE \triangleq 1..10000$

INVARIANT

$stockID\_r = stockID \ \wedge \ stockList\_r = stockList \ \wedge$

$discontinued\_r = discontinued \ \wedge \ price\_r = price \ \wedge$

$priceList\_r = priceList$

INITIALISATION $stockID\_r,\ stockList\_r,$

$discontinued\_r,\ price\_r,\ priceList\_r :=$

$55550,\ \{55550\},\ \textbf{TRUE},\ 1,\ \{55550 \mapsto 1\}$

OPERATIONS

$bb \longleftarrow checkStockList(product, option) \triangleq$

PRE $product \in PRODUCT\_LIST \ \wedge \ option \in OPTION\_LIST$

THEN

IF $product \in stockList\_r$ **THEN**

$bb := \textbf{TRUE}$

ELSE

$bb := \textbf{FALSE}$

END

END;

$cc \longleftarrow checkVendor(product, option) \triangleq$

PRE $product \in PRODUCT\_LIST \ \wedge \ option \in OPTION\_LIST$

THEN

IF $product \in PRODUCT\_LIST$ **THEN**

$cc := \textbf{TRUE}$

ELSE

$cc := \textbf{FALSE}$;

$discontinued\_r := \textbf{TRUE}$

END

END;

$updateStockList(prod) \triangleq$
PRE $product \in stockList\_r \ \wedge \ prod \notin PRODUCT\_LIST$
THEN
$stockList\_r \ := \ stockList\_r - \{prod\}$
END;

$updatePriceList(prod, prc) \triangleq$
PRE $product \in stockList\_r \ \wedge \ prod \in PRODUCT\_LIST \wedge$
$prc \in PRICE\_RANGE$
THEN
$priceList\_r(prod) \ := \ prc$
END
END

In the **INVARIANT** clause of $StaffSalaries\_r$, we defined the additional requirement that the sum of any loan outstanding and any loan being requested must always be a positive natural number (**NAT1**). We show below the listing of $StaffSalaries\_r$ refinement of the $StaffSalaries$ machine.

REFINEMENT $StaffSalaries\_r$
REFINES $StaffSalaries$
SEES $Training\_Mgt, \ PersonnelRecords, \ PerfRelatedPay$
ABSTRACT_VARIABLES $identifier\_r, \ loanStatus\_r,$
$outstandingLoan\_r, \ requestedLoan\_r, \ repaymentAmount\_r,$
$salaryB4Deduction\_r, \ salaryAfter\_r, \ grantLoan\_r$
INVARIANT $identifier\_r = identifier \wedge$
$loanStatus\_r = loanStatus \wedge outstandingLoan\_r = outstandingLoan$
$\wedge \ requestedLoan\_r = requestedLoan \wedge$
$repaymentAmount\_r = repaymentAmount \wedge$
$salaryB4Deduction\_r = salaryB4Deduction \wedge$
$salaryAfter\_r = salaryAfter \ \wedge \ grantLoan\_r = grantLoan \wedge$
$\forall amt \cdot (amt \in \mathbf{NAT1} \wedge outstandingLoan\_r + amt \leq 2147483647 \ \Rightarrow$
$outstandingLoan\_r + amt \ \leq \ 2147483647 \wedge$

$outstandingLoan\_r + amt \;=\; outstandingLoan + amt$)

DEFINITIONS

$EMPID \;\triangleq\; 111100..111199$

INITIALISATION

$identifier\_r,\ loanStatus\_r,\ outstandingLoan\_r,$
$requestedLoan\_r,\ repaymentAmount\_r,\ salaryB4Deduction\_r,$
$salaryAfter\_r,\ grantLoan\_r \;:=$
$\{111100\},\ \{111100 \mapsto NotOwing\},\ 0,\ 0,\ 0\ 0,\ 0,$ **FALSE**

OPERATIONS

$ee, gg \longleftarrow requestLoan(empID) \;\triangleq$

PRE

$empID \in$ **NAT1** $\;\wedge\; empID \in identifier\_r$

THEN

ANY $amount$ **WHERE**

$amount \in$ **NAT** $\;\wedge\; not(amount = 0) \;\wedge$
$outstandingLoan\_r + amount \;\leq\; 2147483647 \;\wedge$
$salaryB4Deduction\_r/4 \;\leq\; 2147483647 \;\wedge$
$0 \;\leq\; salaryB4Deduction\_r/4 \;\wedge$
$0 \;\leq\; outstandingLoan\_r/6 \;\wedge$
$0 \;\leq\; salaryB4Deduction\_r - repaymentAmount\_r$

THEN

IF $outstandingLoan\_r \;=\; 0$ **THEN**
$loanStatus\_r(empID) \;:=\; NotOwing$;
$ee := empID$; $gg :=$ **TRUE**

ELSIF $outstandingLoan\_r \;>\; 0 \;\wedge$
$outstandingLoan\_r + amount \leq salaryB4Deduction\_r/4$

THEN
$loanStatus\_r(empID) \;:=\; Owing$;
$grantLoan\_r \;:=$ **TRUE** ;
$outstandingLoan\_r \;:=\; outstandingLoan\_r + amount$;
$repaymentAmount\_r \;:=\; outstandingLoan\_r/6$;
$salaryAfter\_r \;:=\; salaryB4Deduction\_r -$
$repaymentAmount\_r$; $ee := empID$; $gg :=$ **TRUE**

ELSE
$loanStatus\_r(empID) \;:=\; Owing$;
$grantLoan\_r \;:=$ **FALSE** ;

$salaryAfter\_r := salaryB4Deduction\_r -$
$repaymentAmount\_r$; $ee := empID$; $gg :=$ **FALSE**

END

END

END

END

5.5.2 Redmound-IBM Refinements: Information Flow Analysis

Here in Subsection 5.5.2, we run the formal B refinements defined in Subsection 5.5.1 through our information flow analyser. As noted in Section 5.5, since the local variables defined in the refinements are *linked* via the **INVARIANTS** clause to the variables defined in the respective machine specifications being refined, the security classifications of the *linked* variables must agree. Consequently, without loss of generality, we employ the same security lattice defined in Subsection 5.3.1 and illustrated in Figure 5.20 for the variables native to the refinements presented in Subsection 5.5.1.

Following an approach similar to the one used in Subsection 5.4.2, we begin with a discussion of the results of analysing the refinement of *ManageStock* machine, i.e., *ManageStock_r*. The screenshot showing the output on passing the ascii version of the B refinement *ManageStock_r* and the security policy file *secPol_RIBM.txt* to the flow analyser is presented in Figure 49 in Appendix D. Figure 50 shows our flow analyser adjudged all flows between the variables in operation $bb \longleftarrow checkStockList(product, option)$ to be secure, whereas Figure 51 shows that the flows from *product* into *cc* and *discontinued_r* respectively in $cc \longleftarrow checkVendor(product, option)$ are both insecure, as expected. This is because *product* has a security classification of $[A]$, while both *cc* and *discontinued_r* are public variables with security classification $\varnothing$. And Figure 52 shows that all flows between the variables in operation $updatePriceList(prod, prc)$ are correctly adjudged secure by our flow analyser.

On passing the refinement *StaffSalaries_r* of machine *StaffSalaries* and the security policy file *secPol_RIBM.txt* to our flow analyser, we get the screenshot depicted in 53 of Appendix D. Figures 54 and 56 are relatively straight-

forward since all the flows between the variables are adjudged secure by our information flow analyser. However, Figure 55 shows that the flow from *amount* into *ee* is insecure. This is because *ee* has a security classification of $[C]$ whereas *amount* has security classification $[B]$, hence both variables are incomparable. Similarly *gg* also has a security classification of $[C]$, hence the flow from *amount* into *gg* is correctly adjudged insecure by our security flow analyser. On the other hand, *outstandingLoan_*r, repaymentAmount_*r* and *salaryB4Deduction_r* all have security classification $[ABC]$, hence flows from each of these variables into *ee* (and, similarly, into *gg*), which has a lower security classification of $[C]$, are correctly flagged as insecure by our flow analyser.

Having discussed the B refinements in Redmound-IBM and the information flow analysis thereof, we now shift our focus in Section 5.6 to the B Implementations in the development.

5.6 Redmound-IBM: Implementation

In this section, we present the B Implementations of some of the machines defined in Subsection 5.4.1 above. The implementations have the same number of interfaces and functionality as the corresponding machines/refinements being refined. The B Implementation listings are presented in Subsection 5.6.1, whereas we discuss the results of analysing the implementations with our information flow analyser in Subsection 5.6.2.

5.6.1 Redmound-IBM: B Implementations

We develop, firstly, the *ManageStock* machine through the *ManageStock_r* refinement to the *ManageStock_i* implementation. To build the implementation, however, we have decided to introduce a utility machine *StockDat*[7], which will provide all the operations to be called by *ManageStock_i*. Consequently, *ManageStock_i* EXTENDS *StockDat*. And, secondly, we will be

[7] **Note:** The introduction of *StockDat* machine into the development necessitated some modifications to the earlier versions of *ManageStock* and *ManageStock_r*, but as shown by Figure 5.33, the modified versions satisfy required proof obligations.

implementing the machine *Sales_Technical* straightaway without any intermediate refinement. Figure 5.33 is a screenshot of Atelier B highlighting the consistency checks carried out on the implementations *ManageStock_i* and *Sales_Technical_i*, as well as the machine *StockDat* extended by the *ManageStock_i* implementation.

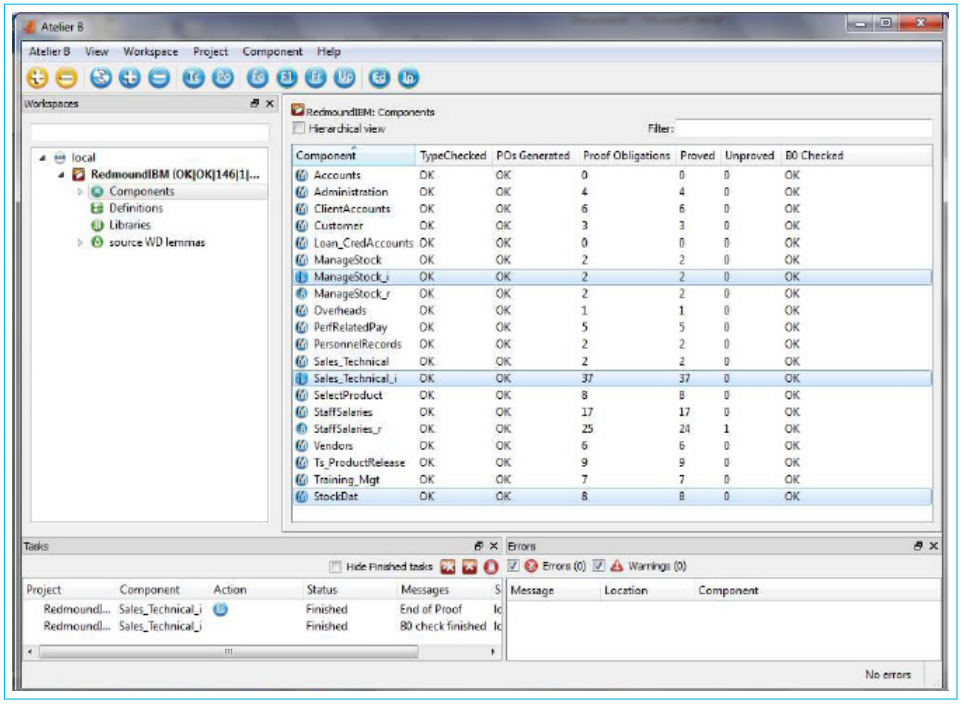


Figure 5.33: *ManageStock_i* : *Consistency Check*

Before presenting the listing of the implementations *ManageStock_i* and *Sales_Technical_i*, we detail below the listing of *StockDat*, which is extended by *ManageStock_i*.

MACHINE *StockDat*
DEFINITIONS
$PRODUCT\_LIST \triangleq 55550..99999$
$PRICE\_RANGE \triangleq 1..10000$
VARIABLES *stockList*, *priceList*, *currentList*
INVARIANT $stockList \subseteq PRODUCT\_LIST \wedge$

$priceList \in stockList \longrightarrow PRICE\_RANGE \wedge$
$currentList \subseteq PRODUCT\_LIST$

INITIALISATION $stockList,\ priceList,\ currentList$
$:= \{55550, 55551\},\ \{55550 \mapsto 1, 55551 \mapsto 10\},\ \{55550\}$

OPERATIONS $\triangleq$

$mp \longleftarrow getPrice\_r(prod)\ \triangleq$
PRE $prod \in PRODUCT\_LIST\ \wedge\ mp \in PRICE\_RANGE$
THEN
$mp\ :=\ priceList(prod)$
END;

$sl \longleftarrow inStockList(prod)\ \triangleq$
PRE $prod \in PRODUCT\_LIST\ \wedge\ prod \in stockList$ **THEN**
$sl\ :=$ **TRUE**
END;

$updateStock(prod)\ \triangleq$
PRE $prod \in PRODUCT\_LIST\ \wedge\ prod \in stockList \wedge$
$prod \notin currentList \wedge stockList - \{prod\} = dom(priceList)$
THEN
$stockList\ :=\ stockList - \{prod\}$
END;

$updatePrice(prod, prc)\ \triangleq$
PRE $prod \in PRODUCT\_LIST\ \wedge\ prod \in stockList \wedge$
$prc \in PRICE\_RANGE$
THEN
$priceList(prod)\ :=\ prc$
END;

$vs \longleftarrow inVendorStock(prod)\ \triangleq$
PRE $prod \in PRODUCT\_LIST$ **THEN**
$vs\ :=$ **TRUE**
END

END

We now present the code listing of the *ManageStock_i* implementation.

IMPLEMENTATION $ManageStock\_i$
REFINES $ManageStock\_r$
SEES $SelectProduct,\ Vendors$
EXTENDS $StockDat$
DEFINITIONS
$PRODUCT\_LIST \triangleq 55550..99999;$
$Red \triangleq 1\ ;\ Blue \triangleq 2;$
$Green \triangleq 3\ ;\ Black \triangleq 4;$
$Pink \triangleq 5$
VALUES
$OPTION\_LIST = Red..Pink$
OPERATIONS
$bb \longleftarrow checkStockList(product, option) \triangleq$
$bb \longleftarrow inStockList(product);$

$cc \longleftarrow checkVendor(product, option) \triangleq$
$cc \longleftarrow inVendorStock(product);$

$updateStockList(prod) \triangleq$
$updateStock(prod);$

$updatePriceList(prod, prc) \triangleq$
$updatePrice(prod, prc)$
END

And, finally, we present the code listing for the *Sales_Technical_i* implementation.

IMPLEMENTATION $Sales\_Technical\_i$
REFINES $Sales\_Technical$
IMPORTS $ManageStock,\ Ts\_ProductRelease$
PROMOTES $checkStockList,\ checkVendor,\ productRelease$
OPERATIONS
$vv \longleftarrow monitorLocalStock(product, option) \triangleq$

$vv \longleftarrow checkStockList(product, option);$

$ww \longleftarrow monitorVendorStock(product, option) \triangleq$
$ww \longleftarrow checkVendor(product, option);$

$uu \longleftarrow monitorProductRelease(pID) \triangleq$
$uu \longleftarrow productRelease(pID)$

END

Having presented the implementation of *ManageStock_i*, we now discuss in subsection 5.6.2 the results of analysing the component with our information flow analyser tool.

5.6.2 Redmound-IBM Implementation: Information Flow Analysis

As in the preceding cases with the B machines (Section 5.4) and their refinements (Section 5.5), we pass the *ManageStock_i* and *Sales_Technical_i* implementations in the B development of Redmound-IBM (Subsection 5.6.1) to our information flow analyser tool to track flows between variables visible within the development. We begin with a discussion of the results of analysing the *ManageStock_i*.

The screenshot in Figure 57 of Appendix D depicts the output on passing the *ManageStock_i* implementation and the security policy file *secPol_RIBM.txt* to our information flow analyser. Figure 58 shows that our flow analyser correctly adjudged flows from $PRODUCT\_LIST$, *prod* and *stockList* respectively into the output variable *bb* within operation
$bb \longleftarrow checkStockList(product, option)$ secure. This is the case because $PRODUCT\_LIST$, *prod* and *stockList* are all public variables with security classification $\varnothing$, whereas *bb* has a security classification of $[A]$ in our security policy file.

Figure 59 shows that all flows within $cc \longleftarrow checkVendor(product, option)$ operation are secure for the reason that all the variables involved are public variables with security classification $\varnothing$. However, in Figure 60, we see that our security flow analyser adjudged the flows from *currentList* and

priceList respectively into *stockList* within *updateStockList*(*product*) to be insecure. This is because both (former) variables are undefined in our security policy file, thus they are incomparable with *stockList*.

And, finally, our analysis of *updatePriceList*(*prod*, *prc*) operation shows that the flows from $PRICE\_RANGE$ and *prc* respectively into *priceList* are adjudged insecure. This is because both $PRICE\_RANGE$ and *prc* have security classification $[A]$ whereas *priceList* is unclassified. However, since the variables $PRODUCT\_LIST$, *prod* and *stockList* are public variables, their flows into *priceList* are appropriately adjudged secure.

On passing the *Sales_Technical_i* implementation coupled with the security policy file *secPol_RIBM.txt* to our information flow analyser, we get the screenshot illustrated in Figure 62. Figure 63 shows that the flow from $VALID\_SID$ into *bb* is insecure for the reason that $VALID\_SID$ is unclassified, whereas *bb* has a security classification of $[A]$. Similarly, since *product* is unclassified, the flow from *product* into *bb*, as shown in Figure 64, is insecure too. For similar reasons, flows from $VALID\_SID$ and *product* (and *discontinued* as shown in Figure 65) respectively into *cc* are adjudged insecure by our security flow analyser (Figure 66). And, finally, Figure 67 shows that the flow from *pID* into *shippingCost* is insecure also.

Having used our information flow analyser tool (developed on the basis of the formal framework introduced in Chapters 3 and 4) to analyse information flow within Redmound-IBM, a conceptual system developed using the B Method, we summarise the benefits of this approach in Section 5.7.

5.7 Benefits of Stepwise Flow Analysis

Integration of our information flow analysis framework into the stepwise software development process is a practical way of applying the theoretic notions of information flow security in the industry in a manner simple and efficient enough to be readily adopted by practitioners.

Bruce Potter in [121] pointed out that rather than addressing the security problem at its heart, which is bad software, "*most defensive mechanisms on*

the market . . . operate in a reactive mode: don't allow packets to this or that port, watch out for files that include this pattern in them, throw partial packets and oversized packets away without looking at them." The information flow analysis framework we proposed in this thesis deals with the problem of software security at its heart by building information flow security into software systems rather than viewing security as an add-on to software systems.

In addition, using our information flow analyser, a developer will be able to review the defined information flow policy to determine if there is a need to modify it, thereby allowing the security classification of variables to *float* up or down, as the case may be, in the manner proposed by Hunt and Sands in [75] (Section 2.2.2.10).

Our information flow analysis framework provides developers with a means to guarantee *only valid and secure refinements are accepted*, since information flow security properties are integrated into the stepwise development process. Hence after checking validity of refinement using existing development tools, before a system is committed at any stage of the development process, it can be automatically checked for secure information flow in all possible interactions between the system variables and other data objects. Whenever it is that a system satisfies the traditional refinement relation while at the same time having secure information flow based on a defined information flow policy, the developer can be assured the system satisfies possibilistic information flow security.

As noted by Sebastian Hunt in [74], [138], a flow-sensitive information flow analysis framework that guarantees end-to-end secure information flow can help with correct formulation of *information erasure policies* on secure information. Giving as example a policy that clients' credit card details be erased from a data controller's system after successful completion of a transaction, the author gave the following psuedocode, which highlights the need for a flow-sensitive information flow analysis framework. It is assumed in this psuedocode fragment that cc denotes credit card details; tt holds the return value of operation $transaction(cc)$ that manipulates clients' credit card information - a return value that we assume leaks some information about cc.

$input\ cc\ from\ user$;
$tt\ := transaction(cc)$;
$output\ tt\ to\ bank$;
$cc\ := 0$

It is obvious that erasing credit card details (cc), by overwriting it with 0, is not sufficient to assure clients that their details have been removed from the data controller's system, since the information has flowed into tt, and tt has not been erased. Depending on the transformation applied to cc by tt, there is the possibility that the supposedly secret input cc could be retrieved from tt by an adversary. The data controller in the scenario painted in this example is clearly in breach of The Data Protection Act 1998 [47], in particular the fifth principle that "*Personal data processed for any purpose or purposes shall not be kept for longer than is necessary for that purpose or those purposes*". An information flow analysis framework, like the one we introduced in this thesis, that overapproximates possibilistic information flow can help to track all variables that the data to be erased has interfered with, and hence *may* leak such secret data. Then an information erasure policy can be defined on the variable(s) to be erased as well as all variables subsequently affected by the processing of the data stored in the original variable(s) to be erased.

Our information flow analysis framework can be readily automated, as demonstrated in this thesis. This reduces the likelihood of errors resulting from manual verification of information flow. Automating the information flow analysis framework also removes the need for high flow-theoretic technical knowhow, since the developer is provided with a *push-button* system that deals with all the information flow analysis logic.

Since verification of information flow security is incrementally conducted at every step of the software development lifecycle, our information flow security framework and secure software development methodology reduces the need for over-reliance on testing at the end of the development.

With our information flow analysis framework and secure software development methodology, possible leaks of secure information can be detected

early on in the development process. This considerably reduces the overall cost of development in comparison with traditional development methods, where such leaks of secure information may go undetected until much later on in the software development process.

In the final chapter of this thesis, we show how we accomplished the aims and objectives set out in Section 1.3. We also discuss other research paths logically derivable from the work done herein.

Chapter 6

Conclusion

We set out to use existing theories on confidentiality properties to formulate an automatable information flow analysis framework to enable developers using the B Method to assure end-to-end secure information flow within their developments. In this chapter we show how this aim of the thesis has been met. We also discuss the research methodology used to get this far, as well as other areas where the work presented in this thesis could be further developed.

6.1 Need for Confidentiality-Preserving Framework

The existence of legislative deterrents like incarceration does not eliminate burglary and other crimes. Similarly, legislation like the Data Protection Act 1998 [47], confidentiality disclaimers, access control mechanisms, and the "penetrate and patch" point solutions to software vulnerabilities are grossly inadequate in assuring confidentiality properties in software systems. Even as a prudent householder takes necessary steps to secure his home against burglars, we have shown that software practitioners need an engineering approach to software development that takes into account confidentiality properties at every step of the development process (see Sections 1.1, 1.2, 2.3.5.5, 5.7). In other words, systematically building constraints, or locks as it were, into the software development process to prevent unwarranted access to secure information within software systems. The research project reported on in this thesis provides such an engineering approach whereby

the theoretic notions of confidentiality properties in the literature are used to develop a framework for analysing information flow within B Machines such that only valid machines and refinements satisfying defined security properties are committed.

A similar point was noted by Benjamin Aziz at the 12th CREST Open Workshop [14], [138] while discussing the limitations of access control systems. He aptly pointed out that "*obligations can be forced on a system, but not on users. You can record the fact that users did not meet obligations and assign penalties, but you cannot really force users to meet obligations*". Hence our approach of integrating information flow security into software systems is much more useful than defining access control constraints on users.

In Section 6.2, we discuss some of the contributions made to the existing body of knowledge on the application of confidentiality properties to a real-world programming language. We also discuss in the following section the limitations of our framework.

6.2 Contributions and Limitations

In Chapter 2, we discussed eighteen different formulations of possibilistic security properties in the literature (Subsection 2.2.2). For the most part, we endeavoured to use the same notation to enhance understanding of the common thread that runs through all these security properties, namely:

> *the initial value of a secret variable may not be read or learned or inferred by an adversary on observation of information available at lower security levels.*

We also discussed in Chapter 2 the notions of programs, specifications and refinement presented in the literature (Section 2.3), including the well-known fact that confidentiality properties are not preserved by the classical refinement relation. We reviewed in Section 2.3.5.4 five existing confidentiality refinement frameworks, with their benefits and limitations presented in Section 2.3.5.5. We presented an introduction to the B Method (Section 2.4), which is the core language used later in our information flow analysis in

Section 3.4. Subsection 2.4.4 dealt with our discussion of five existing security frameworks developed using the B refinement process. In Section 4.3, we presented an example B machine and its refinement, which satisfies the classical refinement relation, but fails to preserve the generalised noninterference property satisfied by the original machine. This is an example of the so-called *refinement paradox* that shows that formalism does not in itself guarantee the preservation of secrecy properties.

In view of the proliferation of '*definitions of security properties*', we did not seek to introduce yet more notions of information flow security. A candid examination of many so-called new notions of security properties reveal that they are simply variants of existing properties like Noninterference. Hence, as described in this thesis, we sought ways of assuring that definitions of security properties expressible as a MLS system hold through the step-wise development process. Chapter 3 presents some of our core accomplishments and contributions in this regard.

In Chapter 3 we extended the flow logic analysis approach in Clark et al, [36], to the abridged Generalised Substitution language of the B Method - abridged in the sense that not all the constructs in the language were considered, rather we focused on the core semantics, sufficient for our analysis to be useful. We developed and proved the correctness of our information flow analysis framework, proving also the model intersection property which guarantees that there is always an acceptable and efficient least analysis for the framework.

Applying our reflective (Action) research methodology, we then reviewed the analysis and presented an optimised method for structuring our information flow analysis. We termed this novel approach, discussed in Section 3.5, *reaching dependencies analysis*. We then presented an example in Section 3.7 to illustrate how this optimised framework could be used to analyse B machines. Using the application of Monotone Frameworks, we showed in Section 3.8 how the analysis can be abstracted away from a specific language semantics into a corresponding input-output framework. (**Note:** The development of an information flow analyser based on monotone frameworks is left for further work, while we focused on the development of a flow analyser

based on our core flow analysis framework presented in Section 3.4.)

A major advantage of our analysis is that although it is formalised on the GSL semantics, it can be easily modified for other language semantics, since many of the constructs, e.g. simple substitution, alternation, etc are generic in semantics though some are syntactically different. Also, separating the problem of refinement from the problem of preservation of confidentiality properties of interest (Section 3.2) makes our approach to the problem of preservation of confidentiality properties through refinement easier than earlier work in the literature, while not sacrificing functionality and scope. Another major advantage of our information flow analysis framework is that an analyser can be developed that can *automatically* analyse B machines for secure information flow at every level of the refinement process, as we showed in Chapter 5. Thus, huge 'expert' labour, time and money can be saved, and the possibility of human error creeping into the analysis process is minimised.

The analysis presented in Chapter 3 deals with information flow between variables in standalone B machines. However, developments are generally structured with multiple machines with various levels of visibility of parts of other machines through *structuring mechanisms* such as SEES, INCLUDES, etc. Hence, we presented in Chapter 4 an extension of the flow analysis framework to track flows resulting due to these structuring mechanisms, thereby giving assurance that secrets are not inadvertently leaked to public variables as a result. The proof-of-concept case-study presented in Chapter 5 illustrates the tracking of intra-machine flows between variables as well as inter-machine flows between variables due to visibility via B structuring mechanisms.

Using C++, we have developed an *automatic* information flow analyser based on the framework introduced in this thesis. We then constructed a case study in Chapter 5, which we analysed for secure information flow using our information flow analyser and an arbitrary non-linear security lattice (Section 5.3). The analysis results discussed in Subsections 5.4.2, 5.5.2, and 5.6.2 illustrate one of the major contributions of this thesis: '*static information flow analysis can be automated to overcome the drudgery and*

potential for human error in manually doing the analysis at every step of the development process, and it is swift, painless and cost-effective.' Information flow analysis at every step of the development process also enhances proper and safe handover of contractual responsibilities between multiple development teams, as each team can be assured their contribution to the development is confidentiality-safe before handover.

One limitation we are aware of with our work is that it is at present only specialised to work with Generalised Noninterference (in the Nondeterministic setting) and Noninterference (in the deterministic setting). However, we are working to extend the approach to other possibilistic security properties such as Nondeducibility, Noninference, etc. The primary advantage of our information flow analysis approach, though, is that once the hard work of developing the necessary framework and proof obligations for a particular security property is complete, developers do not need to manually prove again and again the preservation of security properties at every refinement step. Rather, all that is required is to use the resulting security conditions to automatically check that the security property of interest is preserved at every refinement step. For the developer, using the automated information flow analysis framework is simply a push-button process.

Some may view the extra labour required to define the information flow policy lattice used for the analysis as a downside of analysing information flow at every step of the development lifecycle. We note, however, that even if this extra labour is not incurred at the development phase, similar labour will still be expended at some time after the development, if information flow security is pertinent to the system. It is even as my first supervisor, David Clark, used to say: "*work may be deferred, but cannot be destroyed.*" We opine that the potential *huge* loss in time, effort, cost and reputation (and possibly, *life*) arising from the discovery of security vulnerabilities at a later stage in the software lifecycle warrants the comparatively *minuscule* extra effort, time and cost required for analysing B Machines for information flow at every step of the development process. Hence, doing this *necessary* extra work at the software development phase is a worthwhile tradeoff.

6.2.1 Future Work

We itemise below some areas where the research work presented in this thesis can be further extended.

- Environment-based information flow analysis.
- Extension of analysis framework to other possibilistic security properties in the literature.
- Development of a generic flow analyser (e.g., using the monotone framework).
- Extension of information flow analysis to probabilistic security properties.
- Extension of information flow analysis to capture implicit flows other than those caused by termination behaviour, e.g., implicit runtime flows such as timing flows, buffer overruns, etc.
- Retrenchment of confidentiality properties.

In the remaining part of this section, we discuss our intuitions on the areas, itemised above, wherefor we believe our work can be further extended.

Our intuition on environment-based information flow analysis is that an adversary with access to the text of a B machine or its refinements may be able to deduce something about secure information by studying the logical relationships between variables defined in the **INVARIANT** clause. We conjecture that since all substitutions within a B machine must establish the machine's invariant, the simple fact that a machine is committed assures an adversary with visibility of the text of the machine that the relationships between variables defined in the invariant holds. Since such a deduction is made from studying the *environment* or context in which the machine executes rather than on animations (or runs) of the machine, we term this *environment-based information flow*.

We illustrate this intuition with the following example, which defines a machine $InvDependency$ with static variables xx, yy, where xx holds a secret value passed to it via the operation parameters of $update\_xx(zz)$, and yy is

a public variable while zz is a secret variable. (Note: with a slight abuse of notation, we write '?' in $update\_xx(?)$ to denote the notion that the operation parameter is *secret*.)

MACHINE $InvDependency$
VARIABLES $xx,\ yy$
INVARIANT $xx \in NAT\ \wedge\ yy \in NAT\ \wedge\ xx <= yy$
INITIALISATION $xx, yy := 0, 0$
OPERATIONS

$update\_xx(zz) =$
PRE $zz \in NAT\ \wedge\ zz <= yy$ **THEN**
$xx := zz$
END;

$update\_yy =$
$yy := 7$
END;

$main =$
$update\_yy$;
$update\_xx(?)$
END

END

An information flow analysis of each operation on their own shows that the information flow within each operation is secure. For example, from Table 3.4b, an analysis of $update\_xx(zz)^{\ell}$ gives $\widehat{D}(\ell) \supseteq \{(xx, zz), (xx, yy)\}$, hence flows within the operation are secure. It is obvious that the flow within $update\_yy$ is secure too. Hence the flows within the operation $main$ are also secure. Notice that, due to the order in which the operations are interleaved within $main$, the machine will only be consistent whenever $xx <= yy$, as required by the invariant. Hence an adversary can deduce *indirectly* whether the *secret* stored in xx is less than or equal to 7 by observing whether the machine animates or not. This is so because the fact that the machine animates is sufficient proof to a keen adversary that the machine commits when checked with a tool like the B Toolkit or Atelier B; hence, the adversary can safely conclude that $xx <= 7$.

Our initial investigation of this intuition, which we hope to pursue in the future led us to propose Definition 15, stated below.

Definition 15 (GSL Machine 'INVARIANT' Dependency)**.** *Given that x ranges over the list of variables updated in a machine, M, and that y ranges over the free variables in the invariant, denoted $FV(INV)$, we assert that $classify(y)$ is less than or equal to the lower bound of the security classification function, classify, with respect to all x.*

> Formally, recall that we wrote $\widehat{X}$ to denote the set of variables updated in a substitution. Now, we write $\bigcup_M \widehat{X}$ to denote the union of all variables updated within all substitutions of the machine, M. For all $x \in \bigcup_M \widehat{X}$, we write $min(classify(x))$ to denote the lower bound of the security classification of all $x \in \bigcup_M \widehat{X}$. With these, we define our proposed GSL Machine INVARIANT Dependency for all $y \in FV(INV)$ as follows:
>
> $$\forall y \in FV(INV),\ classify(y)\ \leq\ min(classify(x)).$$

The notion captured in Definition 15 is that to prevent environment-based information flow from a secret variable to a public variable, the security classification of variables used in the **INVARIANT** clause of B machines must not have a higher security classification than any of the variables updated within the machine, otherwise information would indirectly flow from the secret variable x to the *less secure* variable y whenever the machine is animated, notwithstanding whether it terminates or not.

The second area we believe the work in this thesis may be extended involves an extension of our analysis framework to other possibilistic security properties in the literature. As discussed in Section 3.1, our information flow analysis framework is designed to assure satisfaction of Generalized Noninterference, in the nondeterministic case, and correspondingly Noninterference in the deterministic case. Although our framework is reasonably general enough, since most confidentiality properties in the literature can be expressed in terms of Noninterference, we conjecture that our framework can be readily modified to deal with other definitions of secrecy in the literature. We leave the investigation of such extensions for further work.

Yet another possible extension to the work presented in this thesis is the development of a generic flow analyser that is independent of the semantics of the B Method. The information flow analysis method using monotone frameworks presented in Section 3.8 of this thesis may serve as the theoretic foundation for the development of such a generic information flow analyser using C++ or some other programming language. The advantage of such an analyser is great in that it can be used for stepwise end-to-end analysis of information flow notwithstanding whether the system is developed using the B Method, Z, or some other language. This, however, may require the translation of the language into the generic format.

The information flow analysis framework introduced in this thesis is termination sensitive. However, our framework is not guaranteed to prevent runtime implicit flows such as timing leaks, buffer overruns, etc. Further, our analysis framework deals with possibilistic, and not probabilistic information flow. Hence other possible future work of interest include the extension of our analysis to capture other implicit (runtime) flows like timing leaks, buffer overflows, etc., and perhaps, even probabilistic flows of information.

The refinement process entails a notion of correctness with respect to the systematic substitutivity of some concrete system behaviours for some abstract system behaviours until an implementable system is reached [16]. As noted by Banach and Poppleton in [17], [18], [16], and [137], however, developers generally have a better understanding of what the concrete model looks like than how the abstract one should look, thereby necessitating a reverse-engineering of the abstract model from the concrete one. Thus, in reality, developers often experiment with a number of prototypes until a workable and refinable one is realised. To help formalise the process of capturing the abstract model of software systems, [17] introduced the *retrenchment* technique by weakening the strict proof obligations of the refinement process so that all useful models are acceptable even if they cannot be refined to implementation. Hence the proof obligations of retrenchment is more forgiving than the proof obligations of refinement.

As in refinements, a relation (like the linking invariant in B GSL) is defined between the abstract state space and the concrete state space in the retrenchment framework. This relation is termed the *retrieve relation* in [17]. Using this relation together with what the authors termed the *within relation* and the *concedes relation*, [17] introduced a simulation-based formula to define the retrenchment proof obligations. Our intuition is that by constraining either the after-value of the retrieve relation or the after-value of the concedes relation with the security conditions introduced in Section 3.6 of this thesis, it may well be possible to develop a confidentiality-respecting retrenchment technique. In developing such a technique, we conjecture that caution has to be exercised so that the constraining of the retrenchment relation does not end up making the resulting proof obligations so strong that it defeats one of the key aims of retrenchment in the first place, namely: *the weakening of the refinement proof obligations*. We consider this another possible path our research work in this thesis may yet diverge into.

6.3 Conclusion

We have accomplished in this thesis our stated aim to use existing theories on the subject of confidentiality properties to formulate an automatable information flow analysis framework that empowers developers using the B Method to assure end-to-end secure information flow within their developments, and we have demonstrated how such an information flow analyser can be used in a practical way.

Appendix A: References

[**1**] Jean-Raymond Abrial, The B-Book Assigning Programs to Meanings, Cambridge University Press, 1996.

[**2**] Jean-Raymond Abrial, Swiss Federal University (ETH), Zürich, Modeling in Event-B: System and Software Engineering, Cambridge University Press, ISBN: 9780521895569, May 2010.

[**3**] C. Métayer (ClearSy) J.-R. Abrial, L. Voisin (ETH Zürich), Event-B Language, RODIN Deliverable 3.2, Project IST-511599, Public Document, 31st May 2005, http://rodin.cs.ncl.ac.uk.

[**4**] A. Aho, J. Hopcroft, J. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[**5**] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers. Principles, Techniques, and Tools, Second Edition, Pearson Education Inc, 2007.

[**6**] Rajeev Alur, Pavol Černý, and Steve Zdancewic, Preserving Secrecy Under Refinement, University of Pennsylvania, icalp06.

[**7**] Khalid Alzarouni, David Clark, Chunyan Mu, Temitope Onunkun, A Program Analysis Approach to Security Properties under Refinement, 3rd International Workshop on Programming Interference and Dependence, Kongens Lyngby, Denmark, 21 August, 2007.

[**8**] Torben Amtoft and Anindya Banerjee, Information Flow analysis in logical form. In Proceedings of the Eleventh International Static Analysis Symposium (SAS 2004), Verona, Italy, August 2004, volume 3148 of LNCS, pages 100-115. Springer-Verlag, 2004.

[**9**] Juan Bicarregui, Alvaro Arenas, Benjamin Aziz, Philippe Massonet, Christophe Ponsard: Towards Modelling Obligations in Event-B. ABZ 2008: 181-194, 2008.

[**10**] Alvaro Arenas, Benjamin Aziz, Juan Bicarregui, Brian Matthews: Managing Conflicts of Interest in Virtual Organisations. Electr. Notes Theor. Comput. Sci. 197(2): 45-56 (2008).

[**11**] Alvaro E. Arenas, Benjamin Aziz, Juan Bicarregui, Michael D. Wilson: An Event-B Approach to Data Sharing Agreements. IFM 2010: 28-42, 2010.

[**12**] Atelier B User Manual, version 4.0, Clearsy System Engineering, France.

[**13**] Robert Ayres, the Essence of Professional Issues in Computing, Prentice Hall Europe, 1999.

[**14**] Benjamin Aziz, On the use of Event-B in Modeling Data Sharing Agreements, in proceedings of the 12th CREST Open Workshop, Security and Code, Centre for Research on Evolution, Search and Testing, University College London, 6 April 2011.

[**15**] Sara Baase, Computer Algorithms - Introduction to Design and Analysis, Second Edition, Addison-Wesley Publishing Company, 1988.

[**16**] Richard Banach, Retrenchment: An Overview, Lecture notes, Computer Science Department, University of Manchester, United Kingdom. (http://www.cs.man.ac.uk/ banach/retrenchment/tutorial/ Retrenchment.TUTORIAL.pdf, downloaded on 14 April 2011).

[**17**] Richard Banach and Michael Poppleton, Retrenching Partial Requirements into System Definitions: A Simple Feature Interaction Case Study, Requirements Engineering Journal, 8 (2). ISSN 0947-3602, 17 July 2003.

[**18**] Richard Banach and Michael Poppleton, Retrenchment: An Engineering Variation on Refinement. In D. Bert, editor, B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings, LNCS 1393, pages 129147. Springer, 1998.

[19] Michael J. Banks & Jeremy L. Jacob. Unifying Theories of Confidentiality. In 3rd International Symposium on Unifying Theories of Programming, volume 6445 of Lecture Notes in Computer Science, pages 120-136. Springer-Verlag, 2010.

[20] Mike Barnett and K. Rustan M. Leino, Weakest-precondition of unstructured programs, In proceedings of PASTE '05 The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 82-87, New York, NY, USA, 2005. ACM Press.

[21] D. Bell and L. LaPadula, Secure Computer System: Mathematical Foundations and Model, MITRE Corporation Technical Report, M74-244, 1973.

[22] Juan Bicarregui, Operation semantics with read and write frames, In Sixth Refinement Workshop, Workshops in Computer Science, 1994

[23] Juan Bicarregui, Do Not Read This, (CLRC Rutherford Appleton Laboratory, Oxfordshire, UK), In Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right, Springer-Verlag London, UK ï¿½2002, ISBN:3-540-43928-5.

[24] Pierre Bieber and Nora Boulahia-Cuppens, Formal Development of Authentication Protocols, In BCS-FACS Sixth Refinement Workshop, 1994.

[25] Pierre Bieber, Nora Boulahia-Cuppens, Thomas Lehmann, Erich Van Wickeren, Abstract machines for communication security, In Proceedings of IEEE Computer Security Foundations Workshop VI , 1993.

[26] Chiara Bodei, Pierpaolo Degano, Riccardo Focardi, Corrado Priami, Primitives for authentication in process algebras, Theoretical Computer Science, vol 283:2, pages 271-304, 2002.

[27] Chiara Bodei , Pierpaolo Degano , Riccardo Focardi , Corrado Priami, Authentication Primitives for Protocol Specifications, In the Proceedings of the 7th International Conference Parallel Computing Technologies (PaCT 2003), pages 49-65, Novosibirsk, Russia, September 15-19, 2003.

[**28**] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi, Bisimulation and unwinding for Verifying Possibilistic Security Properties, In the Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03), Volume 2575 of LNCS, pages 223-237, Springer-Verlag, 2003.

[**29**] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi, A Proof System for Information Flow Security, Im M. Leuschel, editor, Proceedings of International Workshop on Logic Based Program Development and Transformation, LNCS, Springer-Verlag, 2002.

[**30**] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi, Refinement Operators and Information Flow Security, SEFM 2003, Brisbane, September 22-27, 2003

[**31**] Krysia Broda, Susan Eisenbach, Hessam Khoshnevisan, and Steve Vickers, Reasoned Programming, Prentice Hall International Series In Computer Science, 1994, ISBN:0-13-098831-6.

[**32**] Manfred Broy, Compositional Refinement of Interactive Systems Modelled by Relations*, In proceedings of Lecture Notes in Computer Science, 1998, Volume 1536/1998, 130-149, Springer 1998.

[**33**] Manfred Broy and Ketil Stølen, Specification and Development of Interactive Systems, FOCUS on streams, interface and refinement, Springer, 2001

[**34**] B-Core (UK) Ltd, B-Toolkit User's Manual, release 3.4, 1997

[**35**] C J Burgess, The Role of Formal Methods in Software Engineering Education and Industry, Technical Report: CS-EXT-1995-045, University of Bristol, Bristol, UK, 1995

[**36**] David Clark, Chris Hankin, Sebastian Hunt, Information Flow for Algol-like Languages, Elsevier Science, March 2002

[**37**] David Clark, Sebastian Hunt, Pasquale Malacaria, Noninterference for Weak Observers, Presented at First International Workshop on Programming Language Interference and Dependence (PLID'04), satellite workshop of SAS'04, Verona, Italy, August 2004.

[**38**] Michael Clarkson, Nondeterminism and Information flow, Lecture notes, 10/29/2003

[**39**] Clearsy System Engineering, B Language Reference Manual, version 1.8.6, (c) Clearsy, France

[**40**] http://www.coe.fau.edu/sfcel/define.htm

[**41**] Samuel Colin, Georges Mariano, Vincent Poirriez, A natural extension of B substitutions: postconditions, Foundations of Software Technology and Theoretical Computer Science, 2004

[**42**] Consequence Consortium, Methodologies and tools for data sharing agreement Infrastructure, WP2 Data Sharing Agreement Infrastructure, Secure, dependable and trusted Infrastructures Grant Agreement 214859, Deliverable D2.1, December 2008.

[**43**] The Copyright and Related Rights Regulations 1996, Acts of Parliament, UK, The Stationery Office, ISBN 0110633342
(http://www.patent.gov.uk/copy/legislation/ lawsrelated.htm)

[**44**] Thomas A. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, Second Edition, The MIT Press, Cambridge Massachusetts, 2001.

[**45**] Coverity, Software Integrity for Agile Environments, Copyright ©2010, Coverity, Inc., January 2011.

[**46**] Council of Europe, The European Convention on Human Rights, Rome 4 November 1950. (http://www.hri.org/docs/ECHR50.html)

[**47**] Data Protection Act 1998, Acts of Parliament, UK, The Stationery Office Limited, ISBN 0 10 542998 8
(http://www.opsi.gov.uk/acts/acts1998/19980029.htm)

[**48**] B. A. Davey and H. A. Priestley, Introduction to Lattices and Order, Second Edition, Cambridge University Press, 2002.

[**49**] William S. Davis, Business Systems Analysis and Design, Wadsworth, Belmont, CA,1994.

[**50**] Dorothy E. Denning, A Lattice Model of Secure Information Flow, Communications of the ACM, volume 19, number 5, pages 236-243, May 1976.

[**51**] Dorothy E. Denning and P. Denning, Certification of Programs for Secure Information flow, Communications of the ACM, volume 20, number 7, pp.504-513, 1977.

[**52**] The Digital Millennium Copyright Act of 1998, U.S. Copyright Office Summary, December 1998 (http://www.copyright.gov/legislation/dmca.pdf)

[**53**] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall International, 1976.

[**54**] Steve Dunne, Andy Galloway, and Bill Stoddart, Specification and Refinement in General Correctness, Proceedings of the Third BCS-FACS Northern Formal Methods Workshop, 1998.

[**55**] Anthony Finkelstein and Jeff Kramer, Software engineering: a Roadmap, International Conference on Software Engineering, Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, pages 3 - 22, 2000, ISBN:1-58113-253-0

[**56**] R. Focardi and R. Gorrieri, A Classification of Security Properties, (Extended Abstract), Technical Report UBLCS-93-21, Bologna, Italy, October 1993.

[**57**] R. Focardi and R. Gorrieri, A Classification of Security Properties for Process Algebras, Journal of Computer Security, 3(1): 5-33, 1994/1995

[**58**] Riccardo Focardi and Sabina Rossi, Information Flow Security in Dynamic Contexts, In Proceedings of the IEEE Symposium on Security Foundations Workshop, pages 307-319, IEEE Computer Society Press, 2002.

[**59**] Anup K Ghosh, Chuck Howell, and James A. Whittaker, Building Software Securely from the Ground Up, IEEE Software focus, January / February 2002

[**60**] Tom Gilb and Dorothy Graham. Software Inspection, Wokingham, England: Addison-Wesley, ISBN: 0201631814, 1993.

[**61**] Joseph A. Goguen and José Meseguer, Security Policies and Security Models, Proceedings of the IEEE Symposium on Security and Privacy, pages 11-20, 1982.

[**62**] J A Goguen and J Meseguer, Unwinding and Inference Control, Proceedings of the 1984 Symposium on Security and Privacy, pages 75-86, April 28, 1984.

[**63**] Hazel Hall, Dissertation/Project Hints: Writing Up Your Dissertation/Project, http://www.soc.napier.ac.uk/~hazelh/diss/diss_write.htm, (Last updated by Hazel Hall, 2 June 2008).

[**64**] Chris Hankin, Flemming Nielson, and Hanne Riis Nielson, Principles of Program Analysis, Springer-Verlag (Berlin Heidelberg) New York, Inc. Secaucus, NJ, USA, ISBN:3540654100, 1999.

[**65**] Øystein Haugen and Ketil Stølen, STAIRS - Steps to Analyze Interactions with Refinement Semantics, In the Proceedings of the Sixth International Conference on UML (UML'2003), 2003, LNCS 2863

[**66**] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, UK, Ltd, 1985.

[**67**] International Organization for Standardization. Information technology - Security techniques - Entity authentication mechanism; Part 1: General model. ISO/IEC 9798-1: 2010, Third Edition, JTC 1/SC 27, 16 June 2010.

[**68**] Howard Haughton and Kevin Lano, Specification in B: An Introduction Using The B Toolkit, Imperial College Press, 1996

[**69**] S. Joseph Levine, Writing and Presenting Your Thesis or Dissertation, Michigan State University East Lansing, Michigan USA, http://www.learnerassociates.net/dissthes/, (Last Updated: 03/04/2011 03:26:46).

[**70**] Anany Levitin, Introduction to The Design and Analysis of Algorithms, Second Edition, Pearson International Edition, 2007.

[**71**] C.A.R. Hoare and He Jifeng, Unifying Theories of Programming, Prentice Hall International Series in Computer Science, ISBN-10: 0-13-458761-8, 1998.

[**72**] Eric C. R. Hehner, A Practical Theory of Programming, 2005-3-2 Edition (online), Springer, New York, (first edition, 1993)

[**73**] C A R Hoare, Communicating Sequential Processes, Prentice Hall International UK ltd, London, 1985.

[**74**] Sebastian Hunt, Specification and Enforcement of Information Erasure Policies, in proceedings of the 12th CREST Open Workshop, Security and Code, Centre for Research on Evolution, Search and Testing, University College London, 6 April 2011.

[**75**] Sebastian Hunt and David Sands, On Flow-Sensitive Security Types, POPL'06, January 11-13, 2006, Charleston, South Carolina, USA, 2006.

[**76**] Cynthia Irvine, Geoffrey Smith, and Dennis Volpano, A Sound Type System for Secure Flow Analysis, Journal of Computer Security, IOS Press, pp1-20, 29 July 1996.

[**77**] Jeaneau E., 'Bernard of Chartres', in Dictionary of Scientific Biography, Vol. 3, 19., edited by C. C. Gillispie, (1971)

[**78**] John of Salisbury, The Metalogicon, a twelfth-century defense of the verbal and logical arts of the trivium, translated by Daniel McGarry, Berkeley: University of California Press, 1955.

[**79**] Jeremy Jacob. "Separability and the Detection of Hidden Channels", Information Processing Letters 34(1990) 27-29.

[**80**] Rajeev Joshi (University of Texas, Austin, TX), and K. Rustan M. Leino (DEC SRC, Palo Alto, CA), A Semantic Approach to Secure Information Flow, Proceedings of the Mathematics of Program Construction pages 254-271, 1998.

[**81**] Dale M Johnson and J Todd Wittbold, Information Flow in Nondeterministic Systems, Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 1990, pages 144-161. IEEE Computer Society Press, 1990

[**82**] Jan Jürjens, Secrecy-Preserving Refinement, In Proceedings of FME'01, pages 135-152, 2001.

[**83**] Gregor Kiczales and John Lamping and Anurag Mendhekar and Chris Maeda and Cristina Lopes and Jean-marc Loingtier and John Irwin, Aspect-oriented programming, Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220-242, Springer-Verlag, 1997.

[**84**] E. S. Lee and Aris Zakinthinos, A General Theory of Security Properties, In The Proceedings of the IEEE Symposium on Security and Privacy, pages 94-102, Oakland, CA, 1997

[**85**] Anany V. Levitin, Introduction to the design and analysis of algorithms, 2nd ed., Pearson Addison-Wesley, 2007.

[**86**] Steve Zdancewic, Peng Li, Yun Mao, Information Integrity Policies, University of Pennsylvania, 2003

[**87**] Bob Lubarsky and D.G. Weber, The SDOS project - Verifying Hookup Security, In Third Aerospace Computer Security Conference, pages 7-15. AIAA/ASIS/IEEE, 1987.

[**88**] Michael Luck, How to finish A PhD and have a successful Viva, Kings College London, 2009.

[**89**] Daryl McCullough. "Specifications for Multi-Level Security and a Hook-Up Property," Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy. IEEE Press, May 1987.

[**90**] Daryl McCullough. "Noninterference and the Composability of Security Properties," Proceedings of the 1988 IEEE Symposium on Research in Security and Privacy, pages 177-186. IEEE Press, May 1988.

[**91**] Daryl McCullough, A Hookup Theorem for Multilevel Security, IEEE Transactions on Software Engineering, Vol. 16, No. 6, June 1990

[**92**] John A McDermid, The Theory and Practice of Refinement: Approaches to the Formal Development of Large-Scale Software Systems, based on the proceedings of a Workshop on the Theory and Practice of Refinement, King's Manor, York, UK, 7-8 Jan 1988, Butterworth & Co (Publishers) Ltd, 1989

[**93**] Gary McGraw, Building Secure Software: Why the Standard Approach to Security Doesn't Work, Citigal, Inc, April 3, 2002

[**94**] Gary McGraw and John Viega, Building Secure Software: How to Avoid Security Problems the Right Way, Addison-Wesley Professional, part of the Addison-Wesley Professional Computing Series series, ISBN-10: 0-201-72152-X, Sep 24, 2001.

[**95**] McGraw-Hill Dictionary of Scientific and Technical Terms, 6th edition, published by The McGraw-Hill Companies, Inc., 2003.

[**96**] Annabelle Mclver and Carroll Morgan (Editors), Programming Methodology (Monographs in Computer Science), Springer-Verlag New York Inc., ISBN: 978-0-387-95349-6, 2002.

[**97**] John McLean, "A General Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions", Proceedings of the 1994 IEEE Symposium on Security and Privacy, pages 79-93, IEEE Press, May 1994.

[**98**] John McLean, Whither Noninterference?, Center for High Assurance Computer Systems, U.S. Naval Research Laboratory Washington, D.C. 20375-5337, http://chacs.nrl.navy.mil.

[**99**] Heiko Mantel, Preserving Information Flow Properties under Refinement, proceedings of the IEEE Symposium on Security and Privacy, pages 78-91, California, USA, IEEE Computer Society Press 2001.

[**100**] Heiko Mantel, Unwinding Possibilistic Security Properties, Springer-Verlag, Berlin Heidelberg, 2000.

[**101**] Heiko Mantel, Possibilistic Definitions of Security- an Assembly Kit, In Proceeings of the IEEE Computer Security Foundations Workshop (CSFW'00), pages 185-199, IEEE Computer Society Press, 2000.

[**102**] Petit, D., Poirriez, V., and Mariano, G., 2002a, "Development of Formal Components Using the B Method," Proceedings of the First COLOGNET Joint Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems, number CLIP4/02.0, Facultad de Informatica, Madrid, pp. 35-46

[**103**] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC Press, ISBN: 0-8493-8523-7, Fifth Printing, August 2001.

[**104**] Robert K Merton and Denis Donoghue, On The Shoulders of Giants (OTSOG): A Shandean Postscript, University of Chicago Press, 0226520862, May 1993, Originally published by Free Press in 1965.

[**105**] Alice ter Meulen, Barbara H. Partee, and Robert E. Wall, Mathematical Methods in Linguistics, Kluwer Academic Publishers, 1990.

[**106**] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[**107**] Carroll Morgan, Programming from Specifications, Second Edition, Prentice Hall International, UK, Ltd, October 1998 (first published in 1990)

[**108**] Carroll Morgan, How to brew-up a refinement ordering. In E. Boiten, J. Derrick, and S. Reeves, editors, Proceedings International Refinement Workshop, Eindhoven, volume 259 of ENTCS, pages 123-141, 2009.

[**109**] Carroll Morgan, The shadow knows: Refinement and security in sequential programs. Science of Computer Programming, Volume 74, Issue 8, 1 June 2009, Pages 629-653

[**110**] Joseph M. Morris, Laws of Data Refinement, Acta Informatica 26, 287- 308, Springer-Verlag 1989.

[**111**] Andrew C. Myers and Andrei Sabelfield, Language-based Information Flow Security, IEEE Journal on Selected Areas in Communications, Vol 21, No 1, January 2003.

[**112**] National Security Telecommunications and Information Systems Security Committee, National Information Systems Security (INFOSEC) Glossary, NSTISSI No. 4009, September 2000, National security Agency, Ft Meade, USA.

[**113**] Malcolm Newey, Lecture Notes on Formal Methods for Software Engineering, Weakest Precondition Calculus, Australian National University, 2010.

[**114**] Nielson F. and Nielson H. R., Flow logic and operational semantics. Electronic Notes in Theoretical Computer Science, 10, 1998.

[115] Nielson Hanne Riis and Nielson Flemming, Semantics with Applications - A Formal Introduction, First published 1992 by John Wiley & Sons, revised July 1999 (c)http://www.daimi.au.dk/~hrn.

[116] Colin O'Halloran, "A Calculus of Information Flow", Proceedings of the European Symposium on Research in Computer Security, Toulouse, France, 1990.

[117] The Object Management Group, Business Process Model and Notation (BPMN), Version 2.0, OMG Document Number: formal/2011-01-03 (http://www.omg.org/spec/BPMN/2.0).

[118] Temitope Jos Onunkun, Specifying and Refining Confidentiality using the B-method, 2nd International Workshop on Programming Language Interference and Dependence, Imperial College, London, 6 September, 2006.

[119] Michael Poppleton. Towards Feature-Oriented Specification and Development with Event-B. In Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), volume 4542 of Lecture Notes in Computer Science, pages 367-381. Springer-Verlag, 2007.

[120] Nicolas Stouls, Marie-Laure Potet: Security Policy Enforcement Through Refinement Process CoRR abs/1004.1460: (2010).

[121] Bruce Potter (Booz Allen Hamilton), Editor: Gary McGraw, (cigital.com), Building Security In, Software Security Testing, IEEE Security and Privacy magazine, September-October 2004 (vol. 2 no. 5), pp. 81-85.

[122] Bruno R. Preiss, Data Structures and Algorithms with Object - Oriented Design Patterns in Java, John Wiley and Sons, 2001

[123] Ken Robinson, System Modeling and Design, Lecture notes, University of SouthWales, Australia, October 2005 (http://www.cse.unsw.edu.au/cs2111/)

[**124**] Ken Robinson, System Modelling & Design Using Event-B, School of Computer Science and Engineering, The University of New South Wales, Australia, ©July 2007, Last updated: October 10, 2010 (http://wiki.event-b.org/images/SM%26D-KAR.pdf)

[**125**] Ewa Romanowicz, B Method - An overview through example, McMaster University, Hamilton, Ontario, Canada, April 17 2008.

[**126**] A. W. Roscoe and L. Wulf, Composing and Decomposing Systems under Security Properties, in the proceedings of Eighth IEEE Computer Security Foundations Workshop (CSFW '95), 1995.

[**127**] J. M. Rushby. "The Design and Verification of Secure Systems", ACM Operating Systems Rev. 15(5).

[**128**] Steve Schneider, The B-Method: An Introduction, Palgrave 2001

[**129**] Seehusen, F., Stølen, K., Maintaining information flow security under refinement and transformation, In Formal Aspects in Security and Trust, pp 143-157, Volume 4691 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg, 2007.

[**130**] Geoffrey Smith and Dennis Volpano, A Type-Based Approach to Program Security, In Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE, volume 1214 of Lecture Notes in Computer Science, pages 607-621, Springer, April 1997.

[**131**] David Sutherland, A model of information, In Proceedings of the 9th National Computer Security Conference, pages 175-183, Gaithersburg, Md, Sept. 1986.

[**132**] Christopher Sykes (Editor), No Ordinary Genius - The Illustrated Richard Feynman, 1994.

[133] Stewart Toshach, Best Practices for Data Dictionary Definition and Usage, v. 1.1 2006-11-14 Northwest Environmental Data-Network, 2006 (http://www.pnamp.org/sites/default/files/best_practices_for_data_dictionary_definitions_and_usage_version_1.1_2006-11-14.pdf)

[134] Watchtower Bible and Tract Society of Pensylvania, The Origin Of Life, Five Questions Worth Asking, 2010.

[135] Herbert S. Wilf, Algorithms and Complexity, University of Pennsylvania, Philadelphia, PA 19104-6395, Internet Edition, Summer, 1994, http://www.math.upenn.edu/ wilf/AlgoComp.pdf, viewed 8 December 2009.

[136] http://www.consequence-project.eu/, viewed 7 April 2011.

[137] http://www.cs.man.ac.uk/ banach/retrenchment/, last updated 04 February 2011.

[138] http://crest.cs.ucl.ac.uk/cow/12/, viewed 8 April 2011.

[139] http://en.wikipedia.org/wiki/Big_O_notation, viewed 12 December 2009.

[140] Lars Marius Garshol, BNF and EBNF: What are they and how do they work?, http://www.garshol.priv.no/download/text/bnf.html, viewed 28 January 2012.

[141] http://www.hit.ac.il/staff/leonidm/information-systems/ch25.html #figure25.01 viewed 26 June 2012.

[142] http://msdn.microsoft.com/en-us/library/ms707003(v=vs.85).aspx, viewed 17 April 2010.

[143] http://www.sqa.org.uk/e-learning/ITBDB01CD/page_17.htm, viewed 26 June 2012

[144] http://w2.syronex.com/jmr/edu/db/oracle-data-dictionary, viewed 26 June 2012

[145] http://en.wikipedia.org/wiki/Unifying_Theories_of_Programming, viewed 3 March 2011.

[146] http://en.wikiquote.org/wiki/Richard_Feynman, viewed 21 June 2012.

[147] http://www.yourdon.com/PDF/oldJESA/JESA/JESAchpt10.pdf, viewed 26 June 2012.

Appendix B: Security Typing Rules

(IDENT) $\lambda;\gamma \vdash x : \tau \qquad \gamma(x) = \tau$

(VAR) $\lambda;\gamma \vdash x : \tau\ var \qquad \gamma(x) = \tau\ var$

(ACCEPTOR) $\lambda;\gamma \vdash x : \tau\ acc \qquad \gamma(x) = \tau\ acc$

(VARLOC) $\lambda;\gamma \vdash l : \tau\ var \qquad \lambda(l) = \tau$

(INT) $\lambda;\gamma \vdash n : \tau$

(R-VAL) $$\frac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash e : \tau}$$

(L-VAL) $$\frac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash e : \tau\ acc}$$

(SUM) $$\frac{\lambda;\gamma \vdash e : \tau,\ \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e + e' : \tau}$$

(COMPOSE) $$\frac{\lambda;\gamma \vdash c : \tau\ cmd,\ \lambda;\gamma \vdash c' : \tau\ cmd}{\lambda;\gamma \vdash c; c' : \tau\ cmd}$$

(LETVAR) $$\frac{\lambda;\gamma \vdash e : \tau,\ \lambda;\gamma[x : \tau\ var] \vdash c : \tau'\ cmd}{\lambda;\gamma \vdash \textbf{letvar}\ x := e\ \textbf{in}\ c : \tau'\ cmd}$$

(ASSIGN) $$\frac{\lambda;\gamma \vdash e : \tau\ acc,\ \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e := e' : \tau\ cmd}$$

(IF) $$\frac{\lambda;\gamma \vdash e : \tau,\ \lambda;\gamma \vdash c : \tau\ cmd,\ \lambda;\gamma \vdash c' : \tau\ cmd}{\lambda;\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau\ cmd}$$

(WHILE) $$\frac{\lambda;\gamma \vdash e : \tau,\ \lambda;\gamma \vdash c : \tau\ cmd}{\lambda;\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd}$$

(PROCEDURE) $$\frac{\lambda;\gamma[x_1 : \tau_1,\ x_2 : \tau_2\ var,\ x_3 : \tau_3\ acc] \vdash c : \tau\ cmd}{\lambda;\gamma \vdash \textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c : \tau\ proc(\tau_1, \tau_2\ var, \tau_3\ acc)}$$

(APPLY) $$\frac{\lambda;\gamma \vdash e : \tau\ proc(\tau_1,\ \tau_2\ var,\ \tau_3\ acc),\quad \lambda;\gamma \vdash e_1 : \tau_1,\ \lambda;\gamma \vdash e_2 : \tau_2\ var,\ \lambda;\gamma \vdash e_3 : \tau_3\ acc}{\lambda;\gamma \vdash e(e_1, e_2, e_3) : \tau\ cmd}$$

(LETPROC) $$\frac{\lambda;\gamma \vdash \textbf{proc}(\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c : \pi,\quad \lambda;\gamma \vdash [\textbf{proc}(\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c/x]c' : \tau\ cmd}{\lambda;\gamma \vdash \textbf{letproc}\ x(\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c\ \textbf{in}\ c' : \tau\ cmd}$$

Summary of Volpano and Smith's [130], [76] Security Typing Rules.

Appendix C: Software Tools and IDEs Used in Research

i. BizAgi: BPMN Process Modeller (http://www.bizagi.com/).

ii. Microsoft Visio Professional 2010: Advanced diagramming tool.

iii. Atelier B : An industrial tool Developed by ClearSy, to efficiently develop defect-free proven Software using the B Method (http://www.atelierb.eu/en/).

iv. Code::Blocks: Cross-platform C++ Integrated Development Environment (IDE) (http://www.codeblocks.org/).

v. Microsoft Windows Visual C++ 2008: C++ IDE.

vi. Microsoft Windows Visual C++ 2010: C++ IDE.

vii. Microsoft Word 2010.

viii. MiKTeX 2.9: A typesetting system for Windows (www.miktex.org).

ix. WinEDT 7.0: Text (LaTeX) editor for Windows (http://www.winedt.com/).

x. LyX: A document processor (http://www.lyx.org/).

xi. TpX: A drawing tool for TeX (http://sourceforge.net/projects/tpx/).

xii. InstallShield: Microsoft Windows Software Installer (http://www.flexerasoftware.com/products/installshield.htm).

xiii. Adobe Acrobat 2007 .

xiv. Microsoft Windows 7 Paint.

Appendix D: Flow Analyser Screenshots: B Machines

Screenshots of information flow analyser outputs.

Flow analysis of Redmound-IBM Specifications

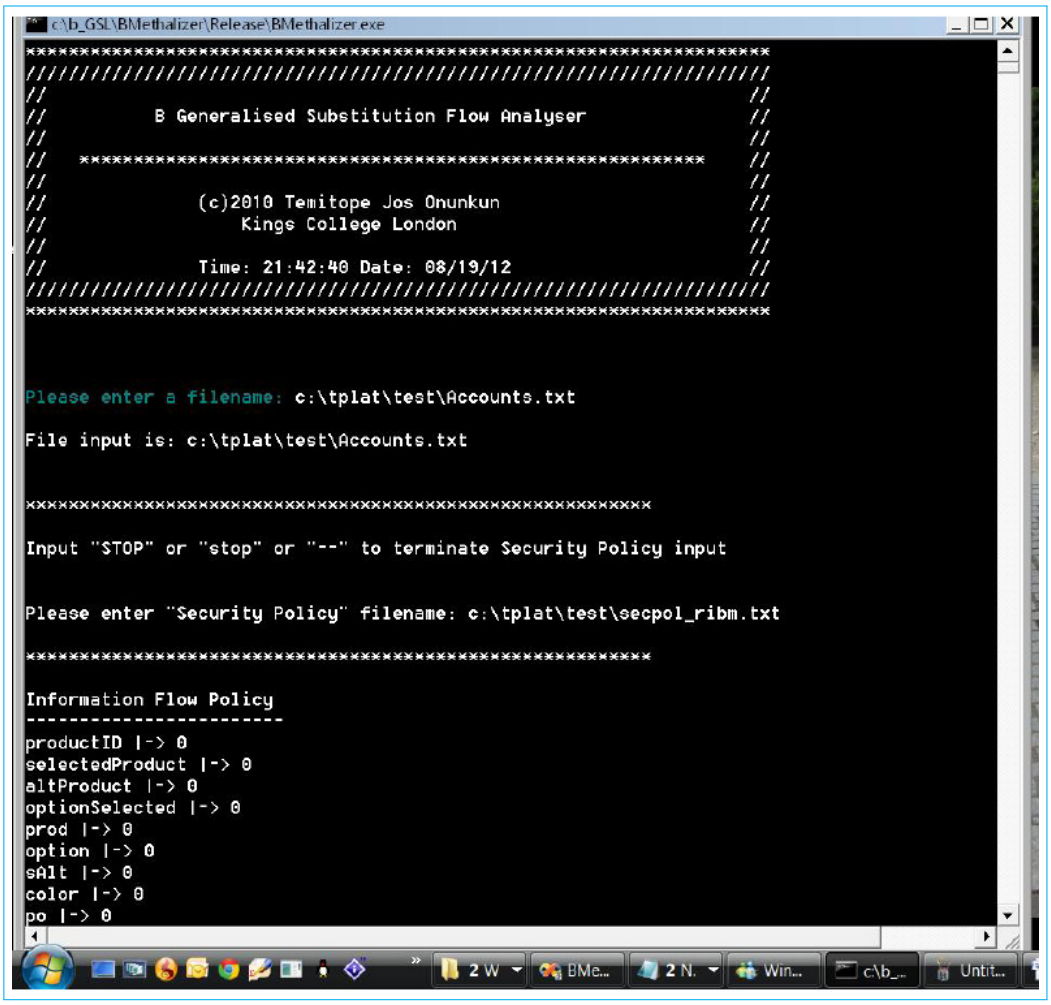


Figure 1: Redmound-IBM: MACHINE *Accounts* : *Input Files*

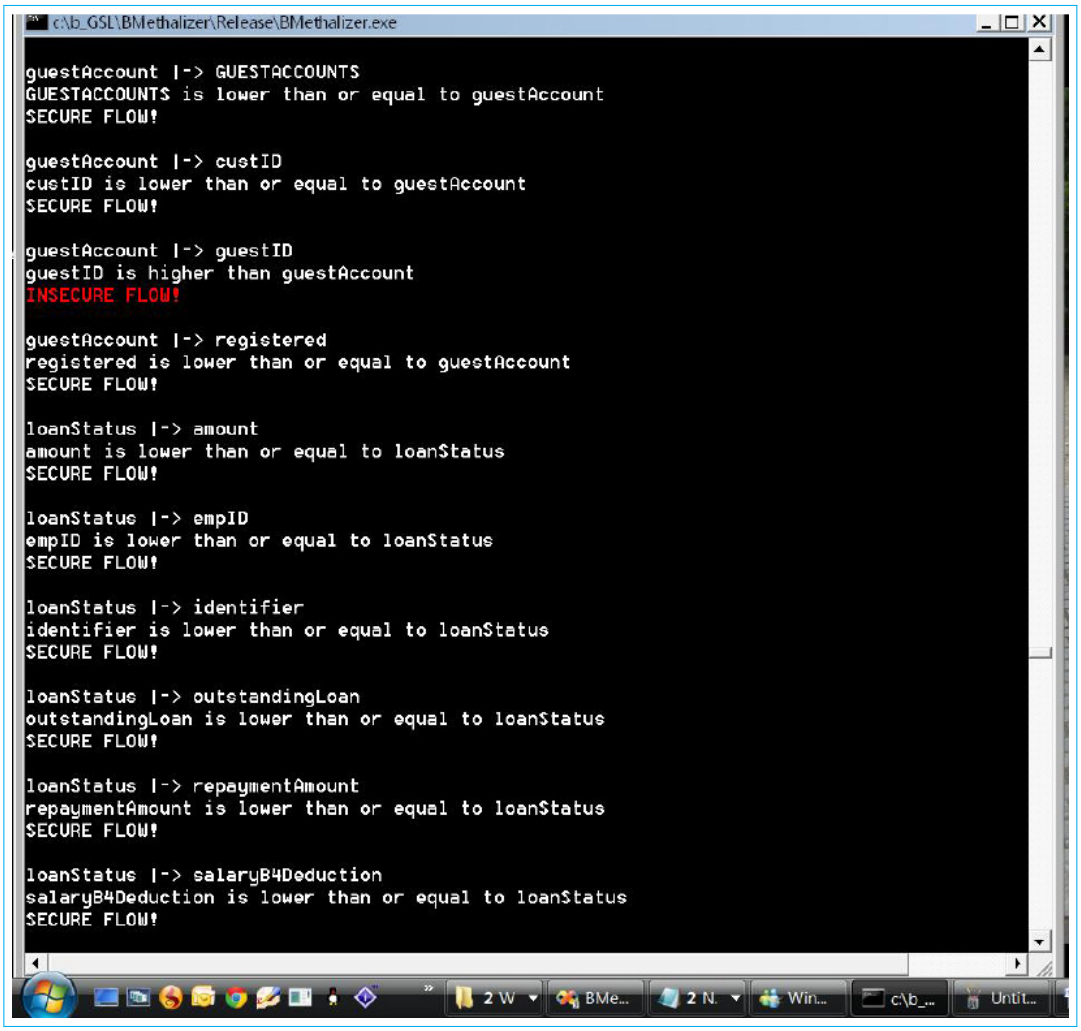


Figure 2: MACHINE *Accounts* : *Flow Analysis*

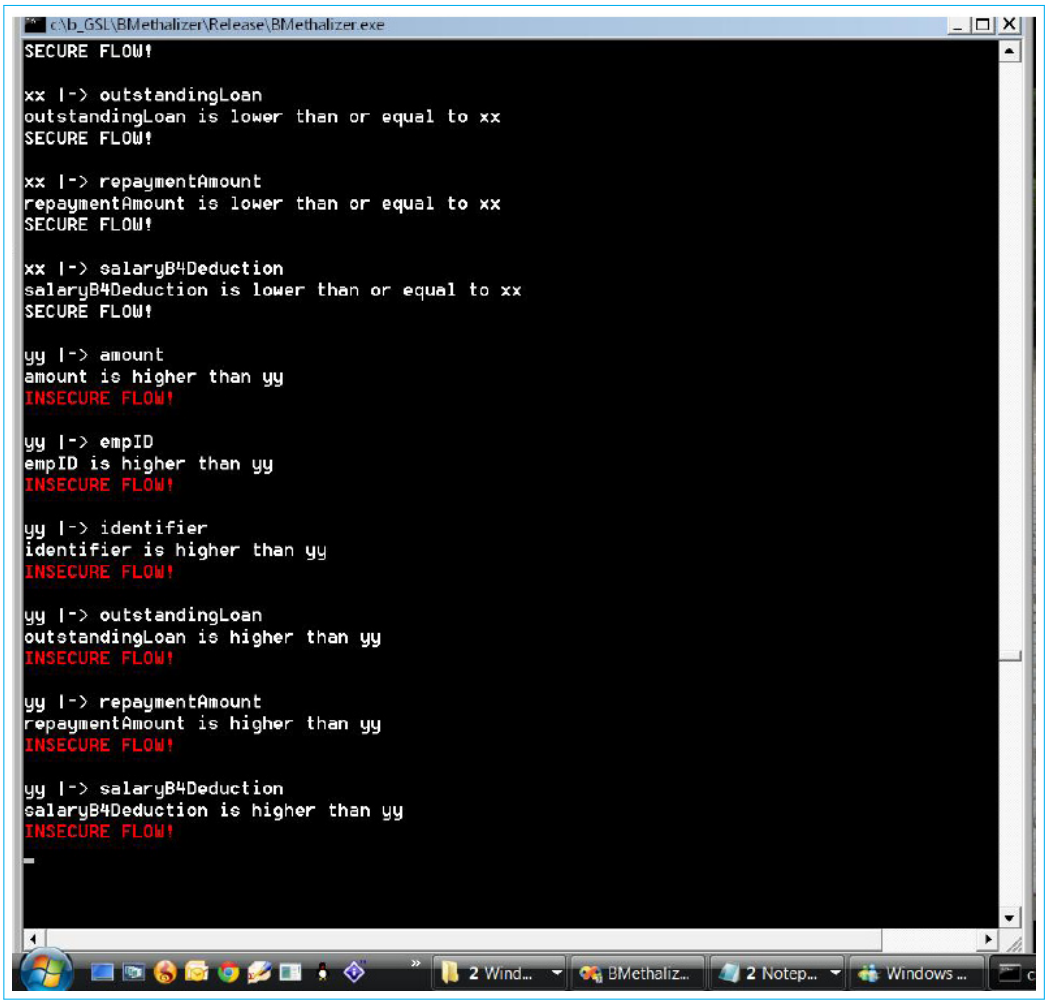


Figure 3: MACHINE *Accounts* : *Flow Analysis*

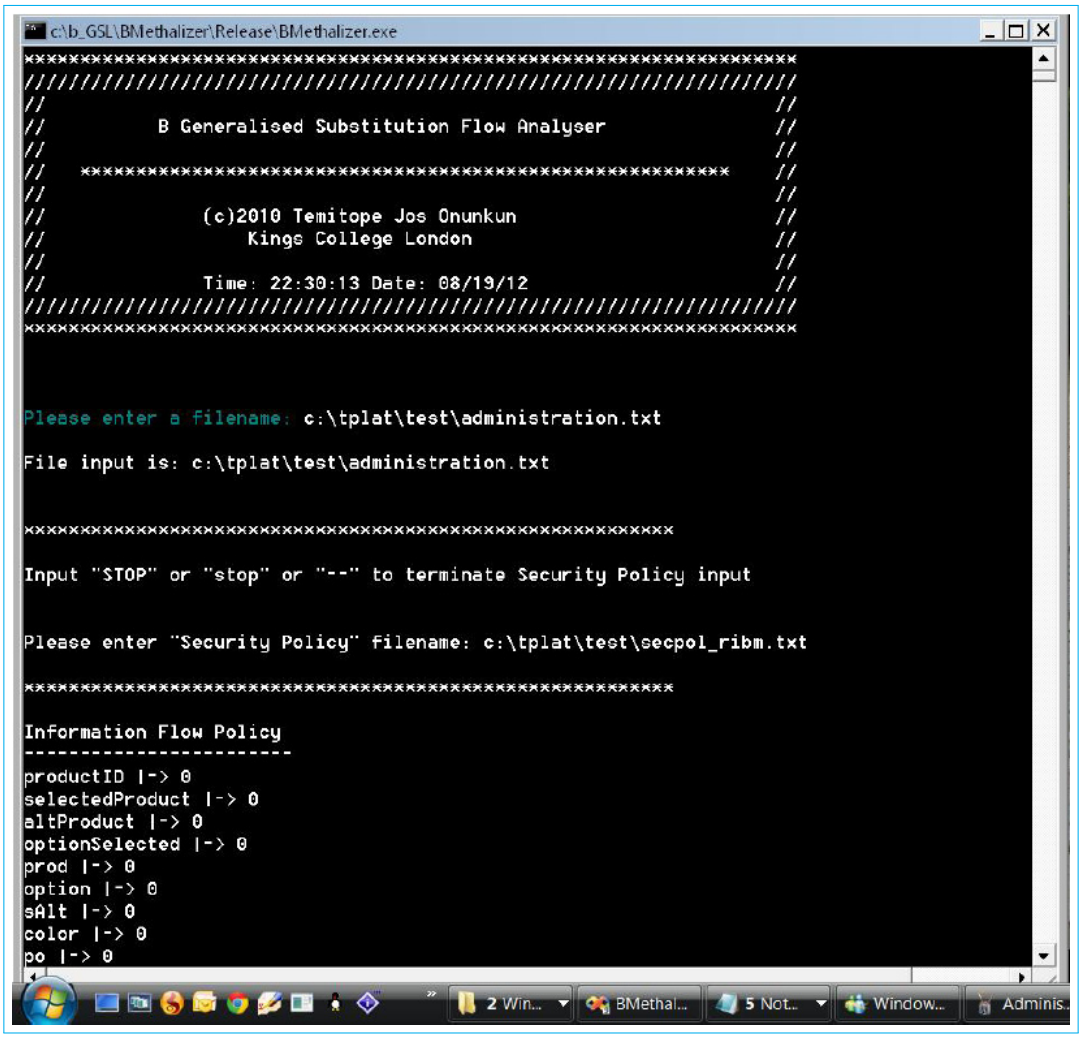


Figure 4: MACHINE *Administration* : *Input Files*

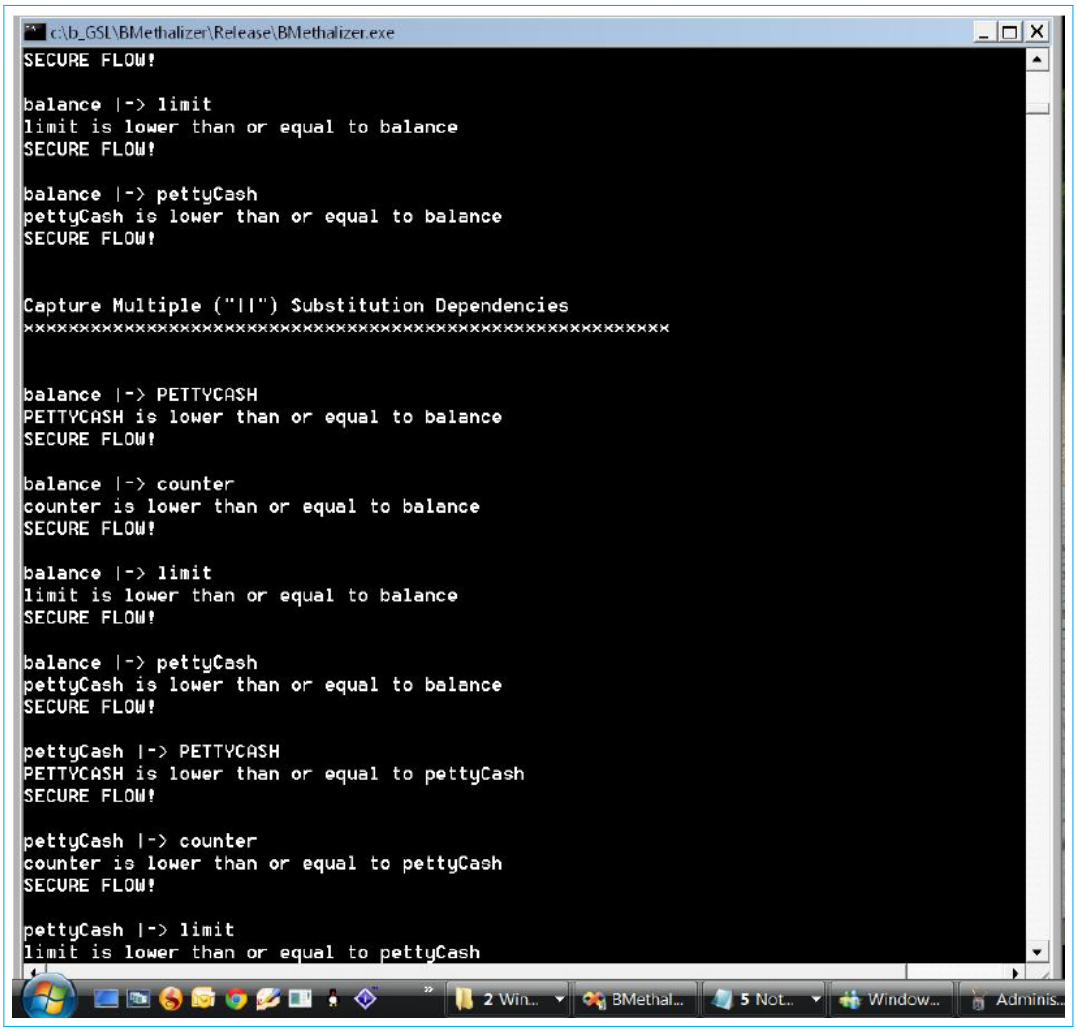


Figure 5: MACHINE *Administration* : *Flow Analysis*

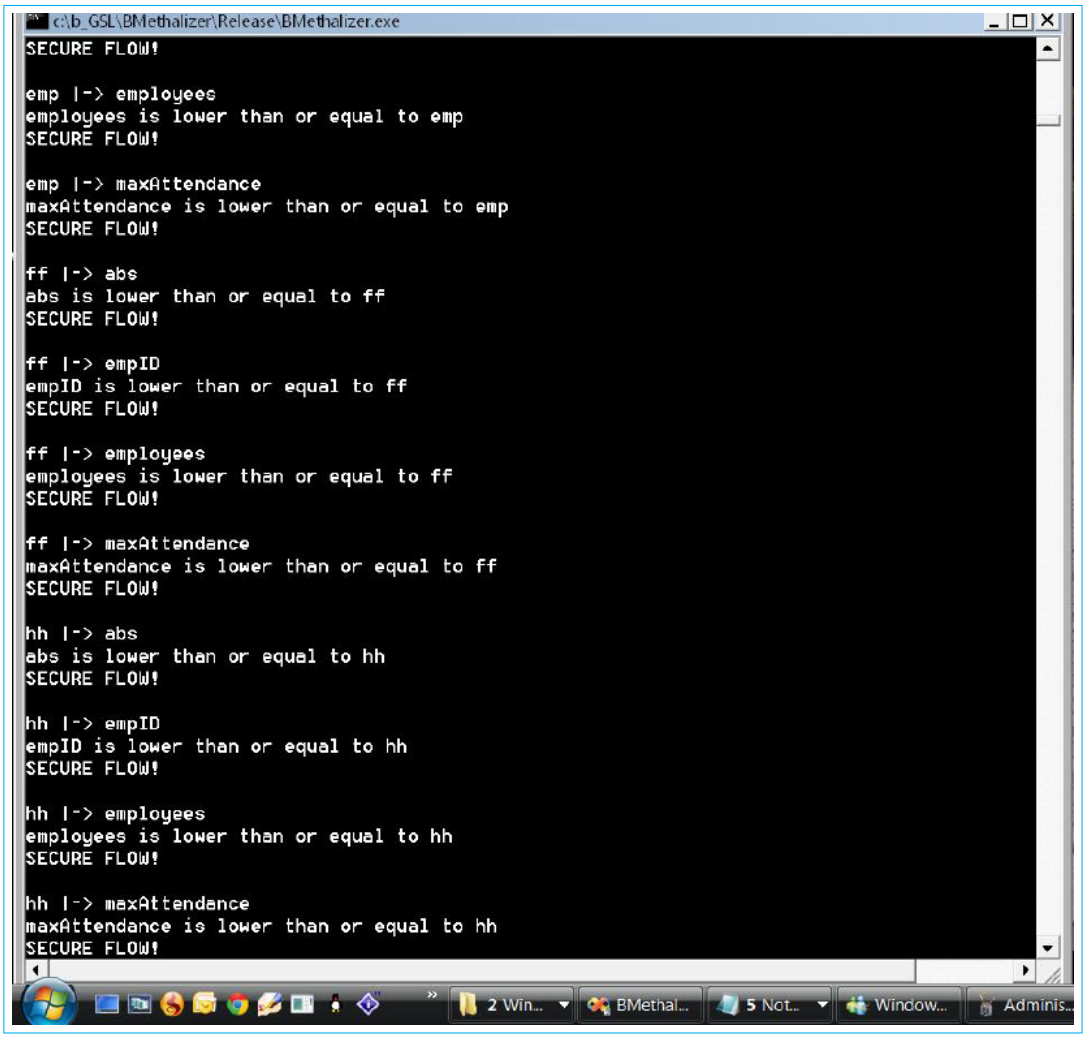


Figure 6: MACHINE *Administration* : *Flow Analysis*

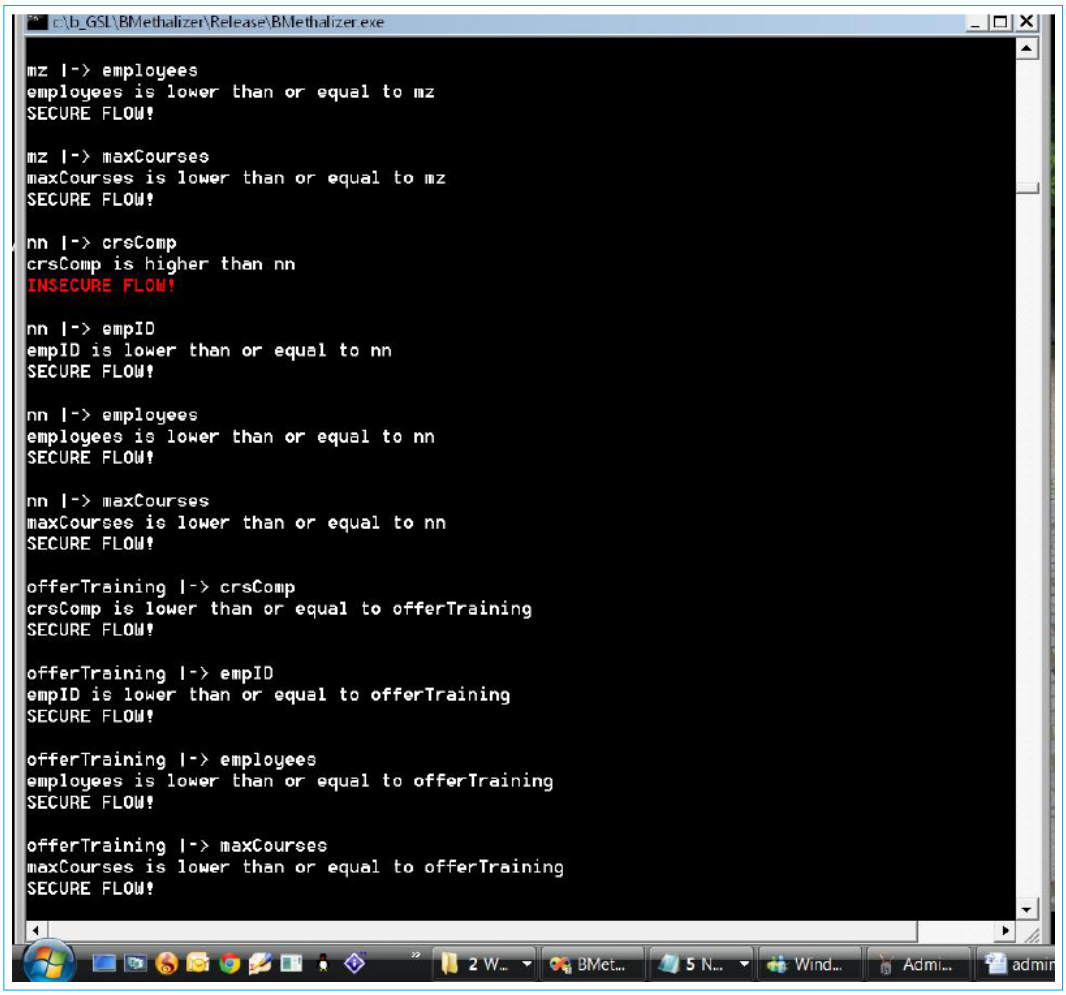


Figure 7: MACHINE *Administration* : *Flow Analysis*

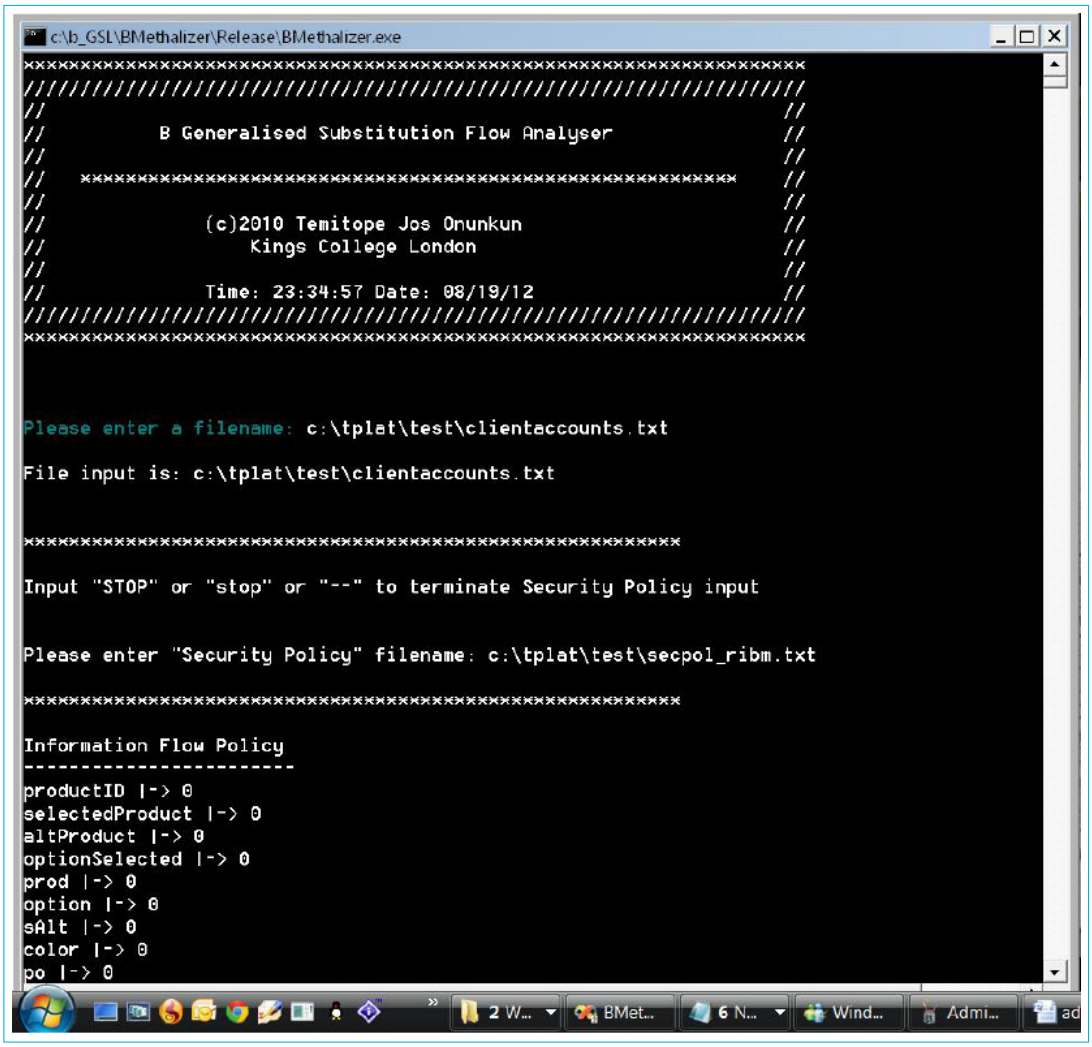


Figure 8: Redmound-IBM: MACHINE *ClientAccounts Input Files*

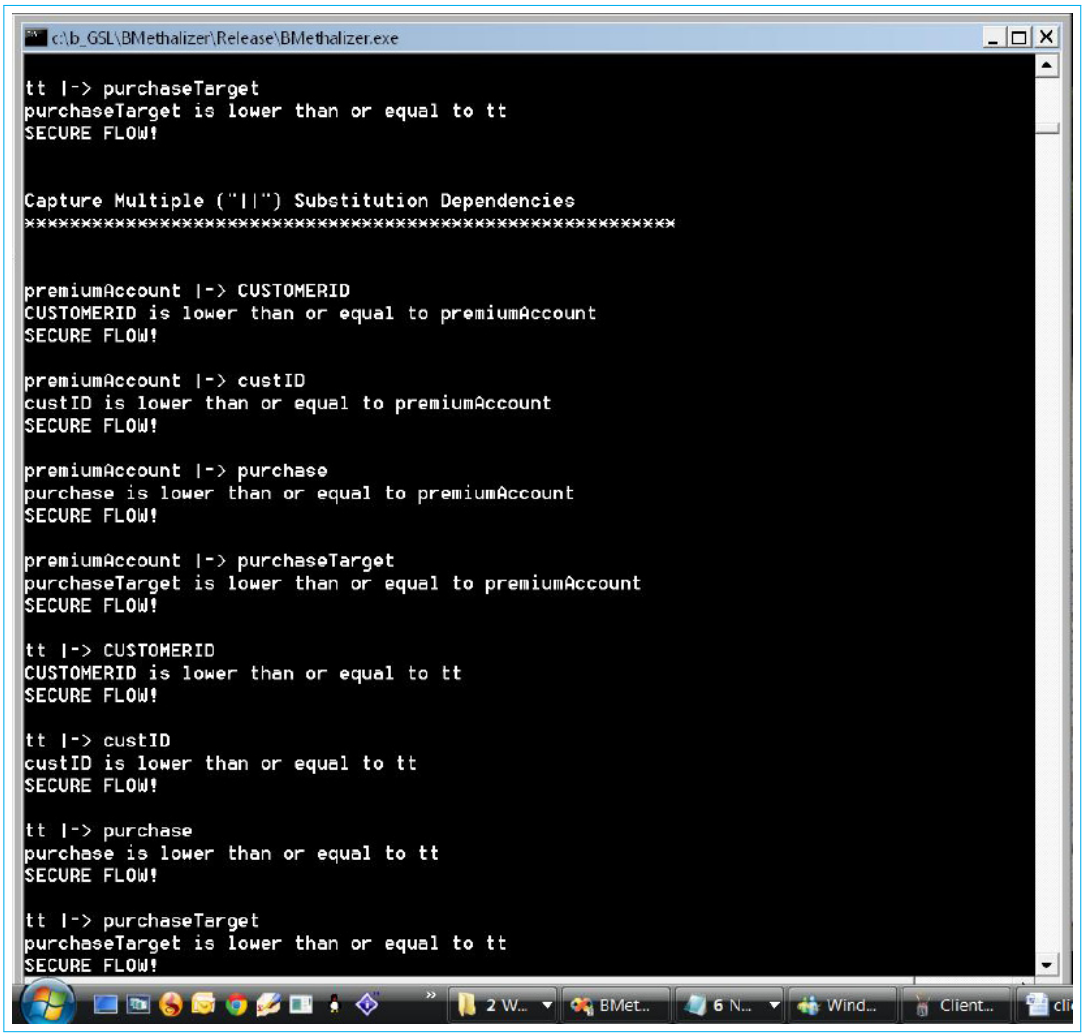


Figure 9: MACHINE *ClientAccounts* : *Flow Analysis*

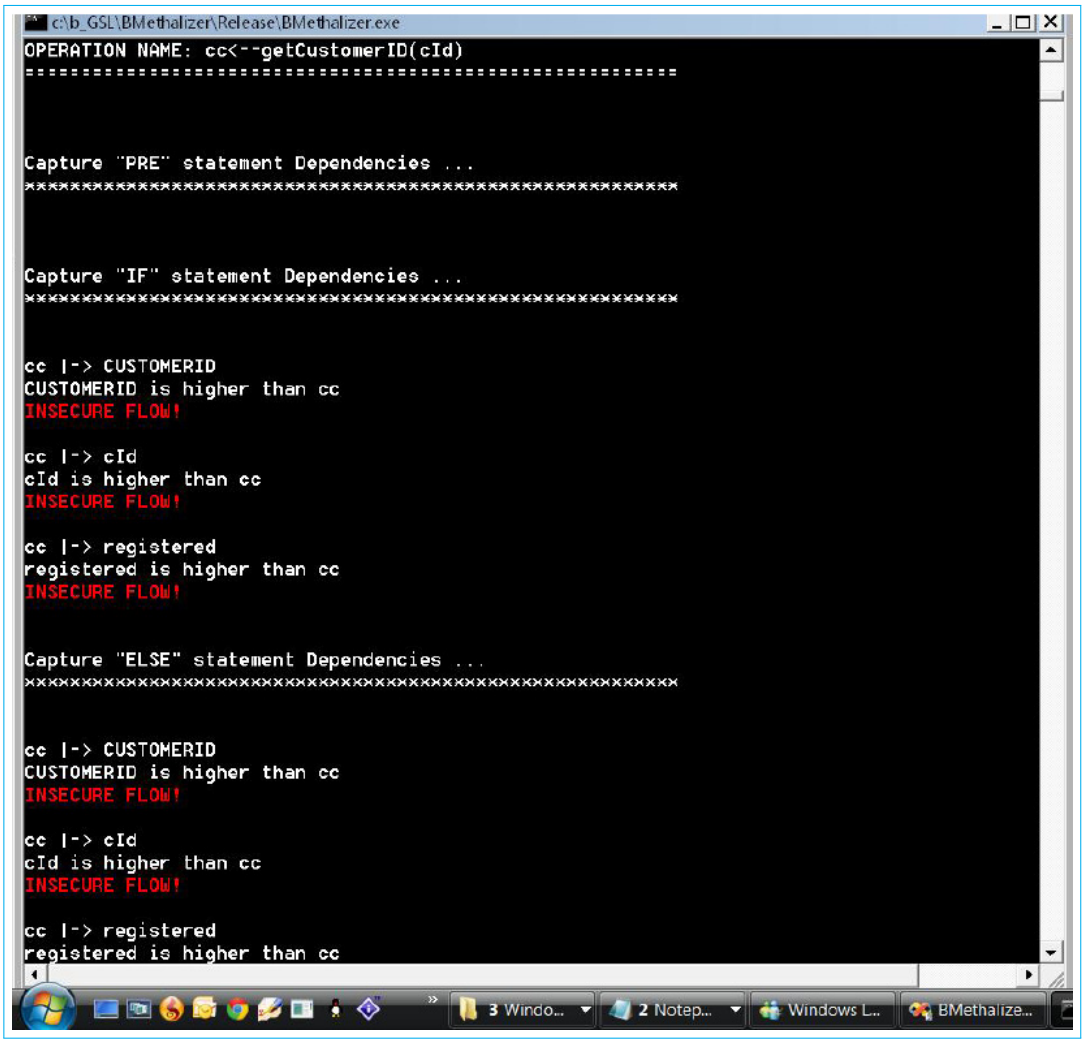


Figure 10: MACHINE *ClientAccounts* : *Flow Analysis*

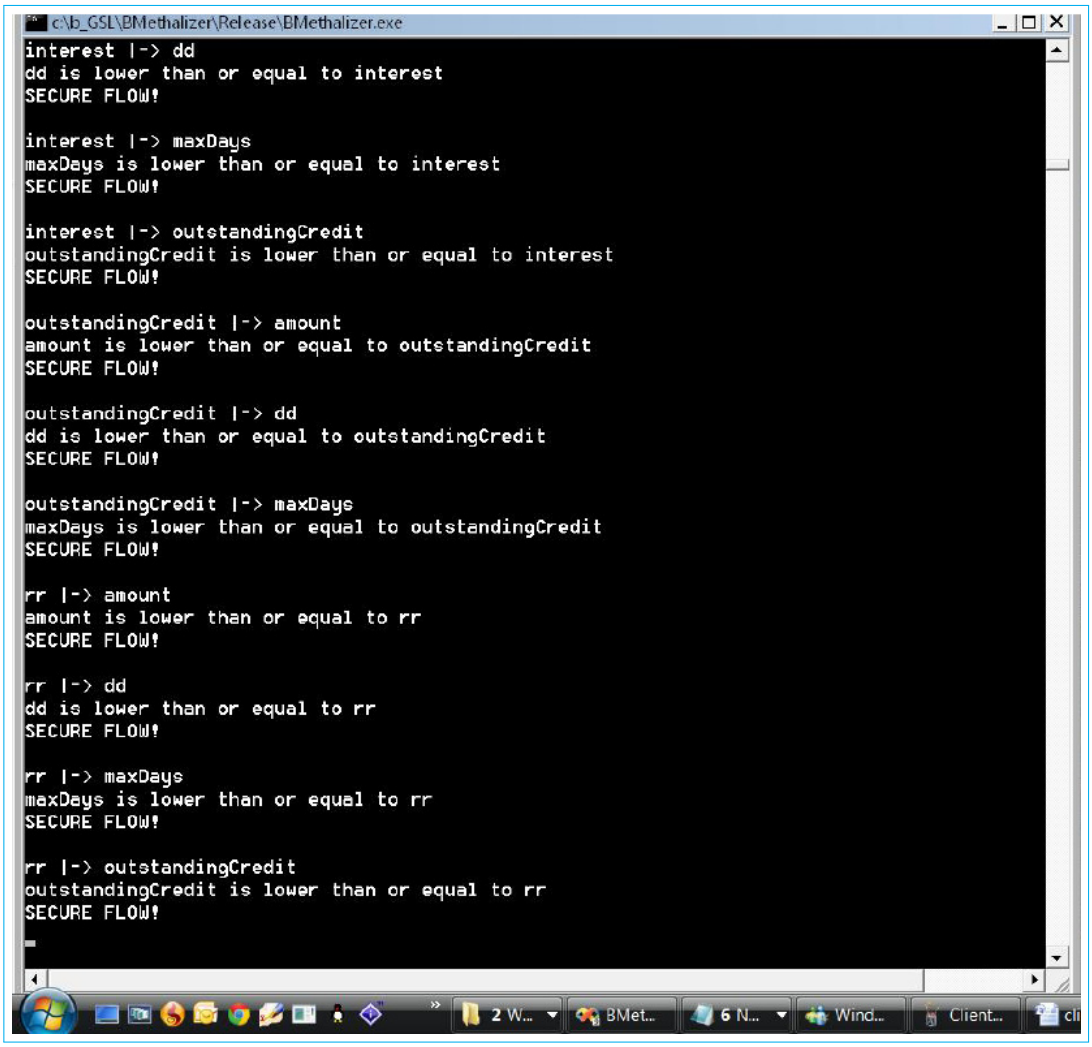


Figure 11: MACHINE *ClientAccounts* : *Flow Analysis*

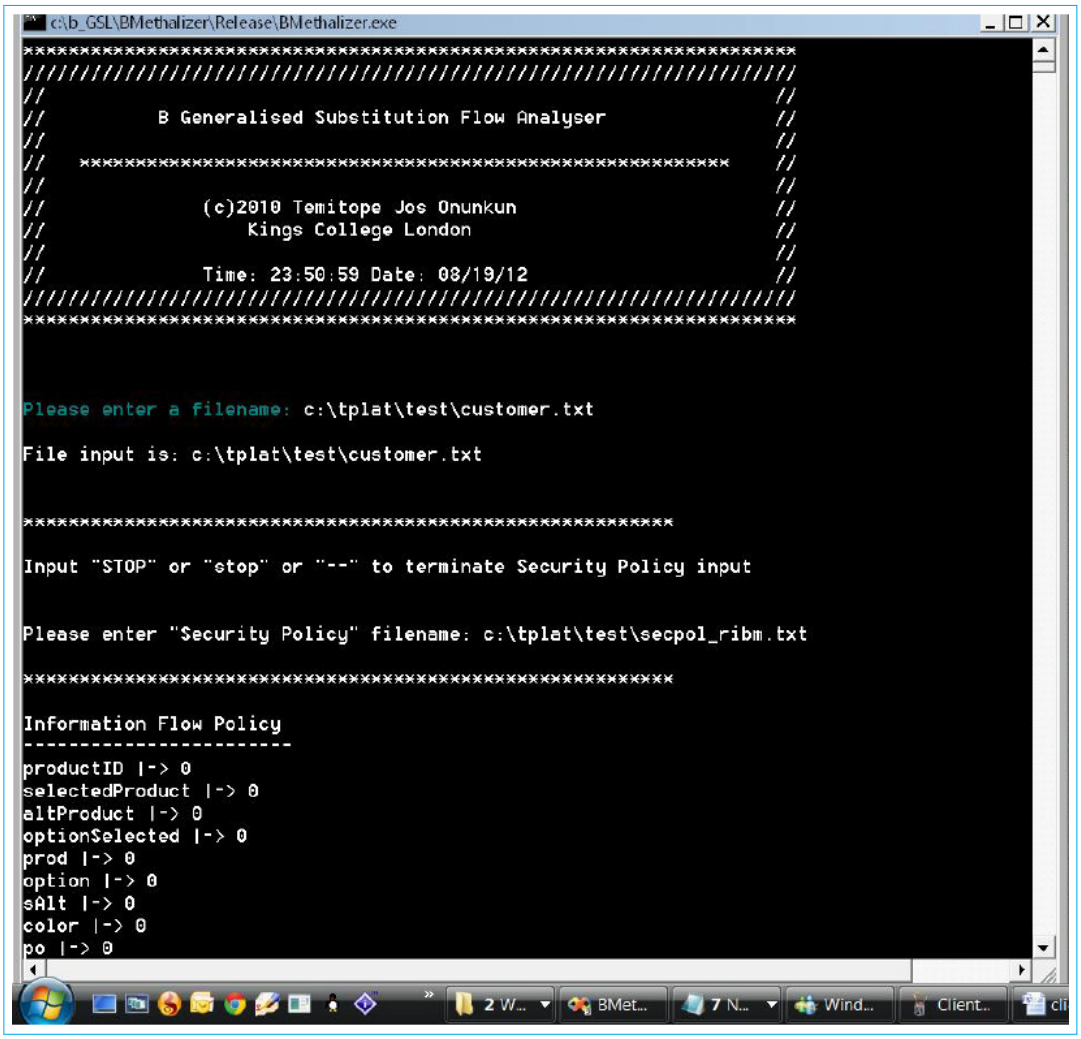


Figure 12: MACHINE *Customer Input Files*

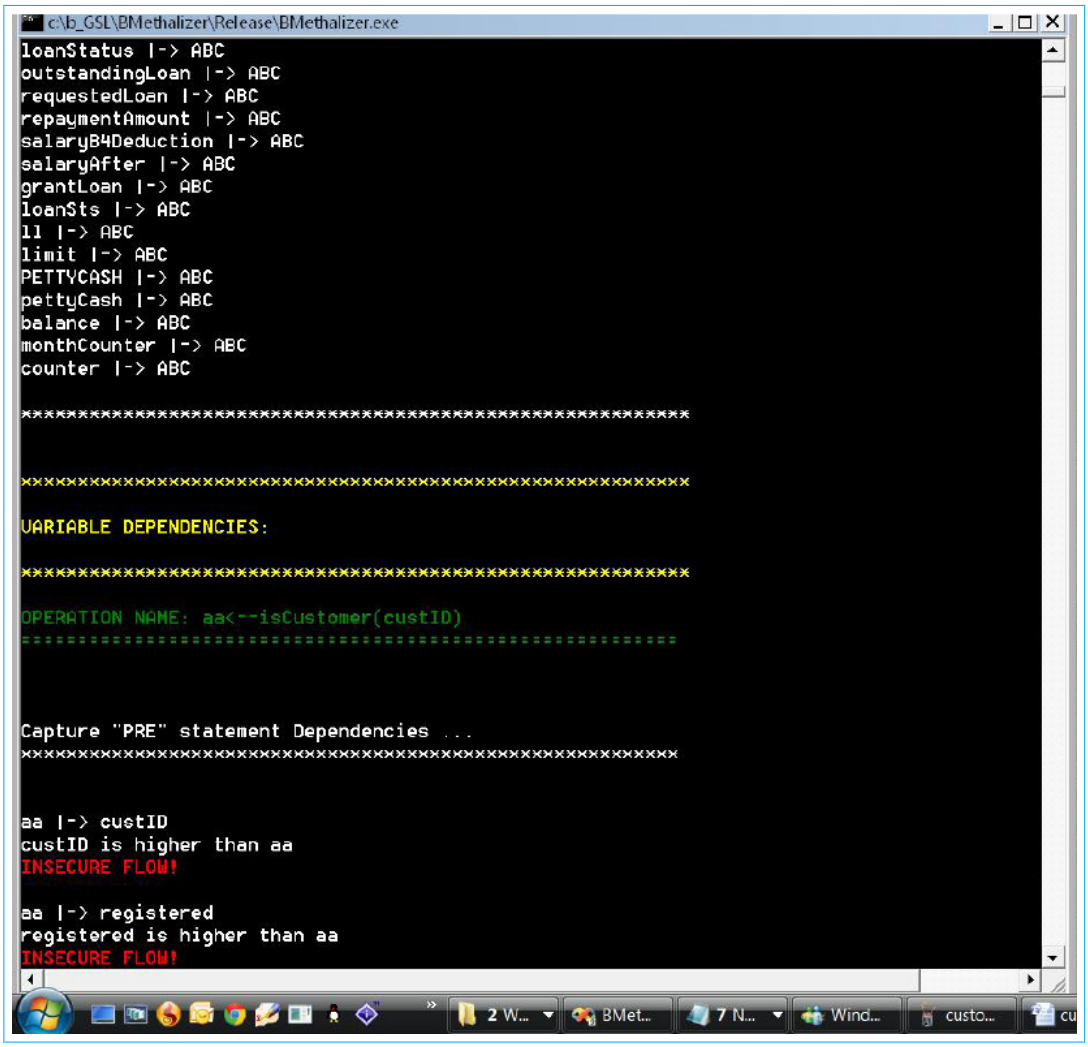


Figure 13: *Customer* : *Flow Analysis*

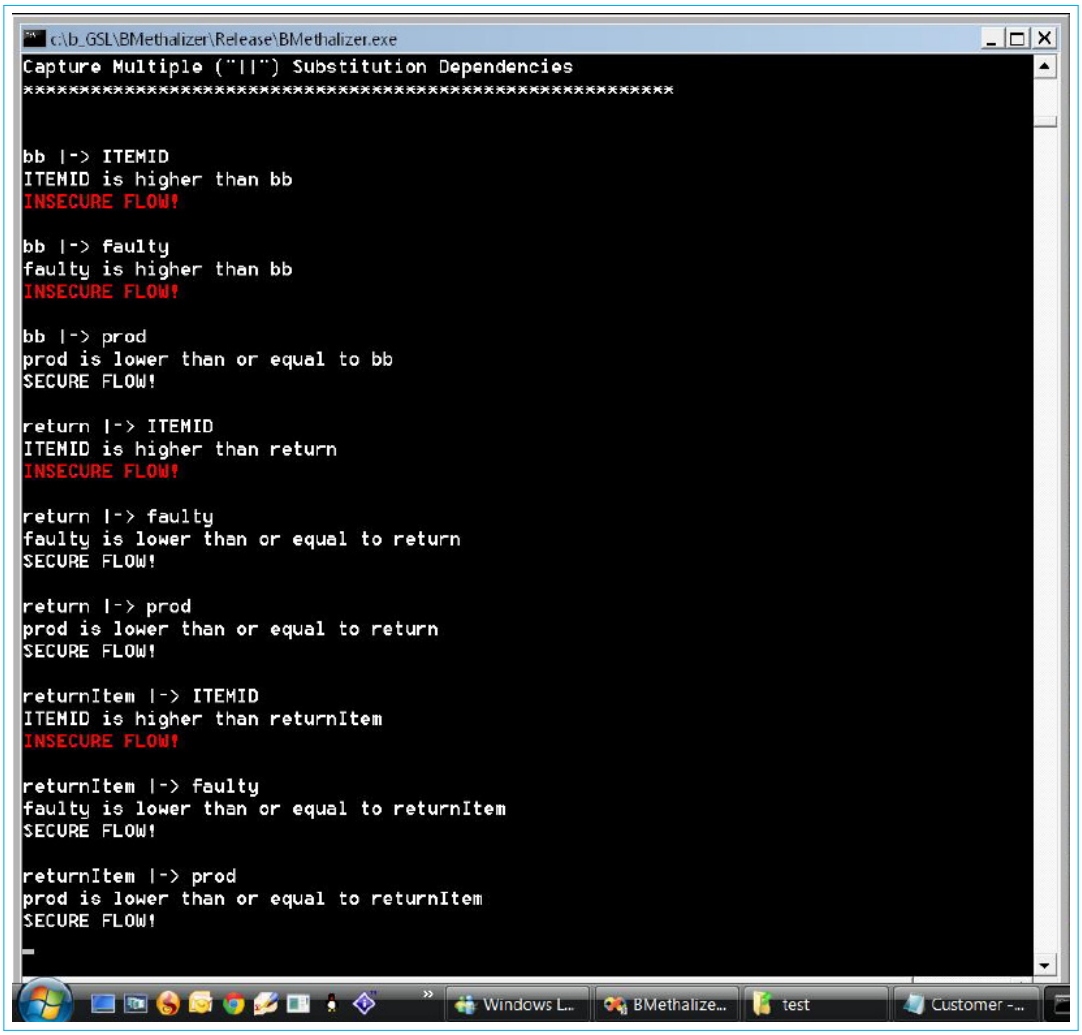


Figure 14: *Customer* : *Flow Analysis*

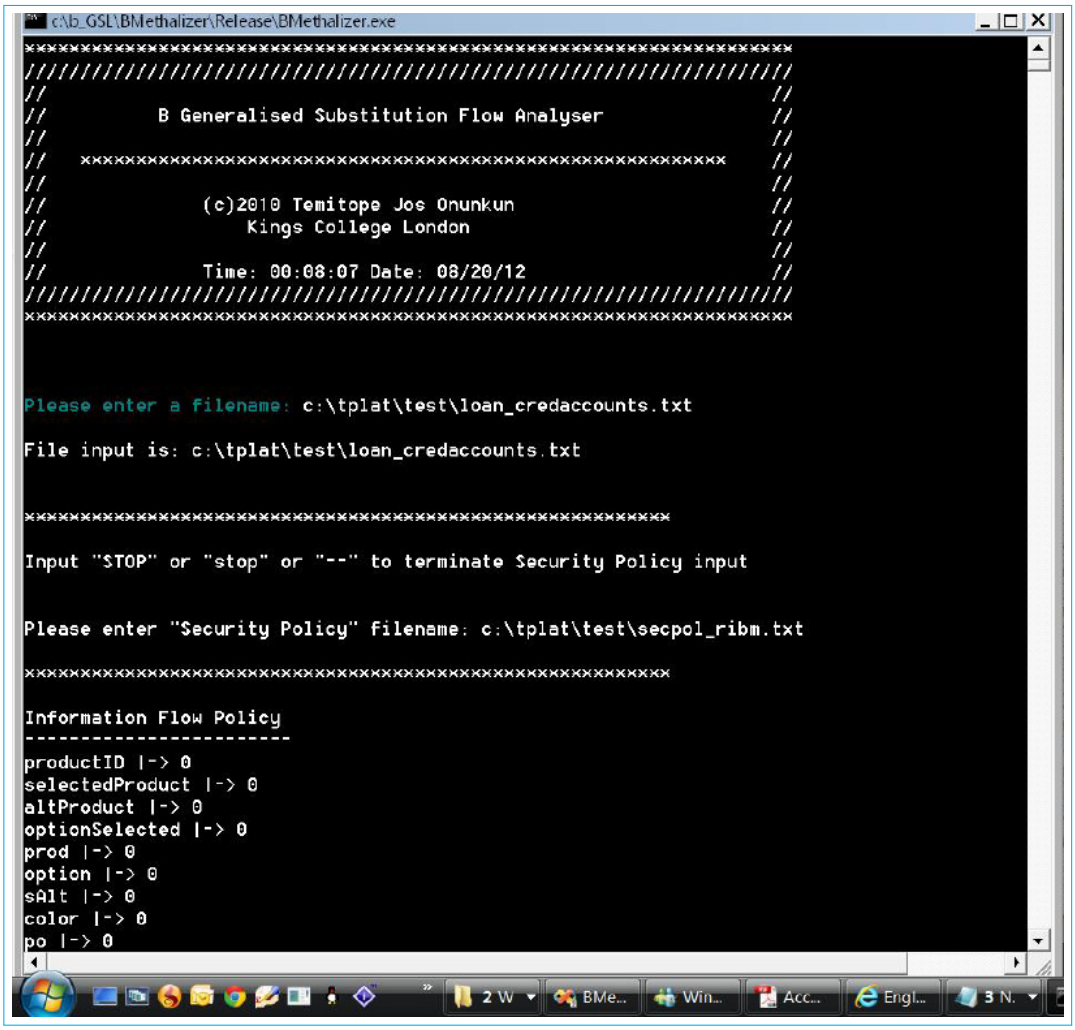


Figure 15: *Loan_CredAccounts* : *Input Files*

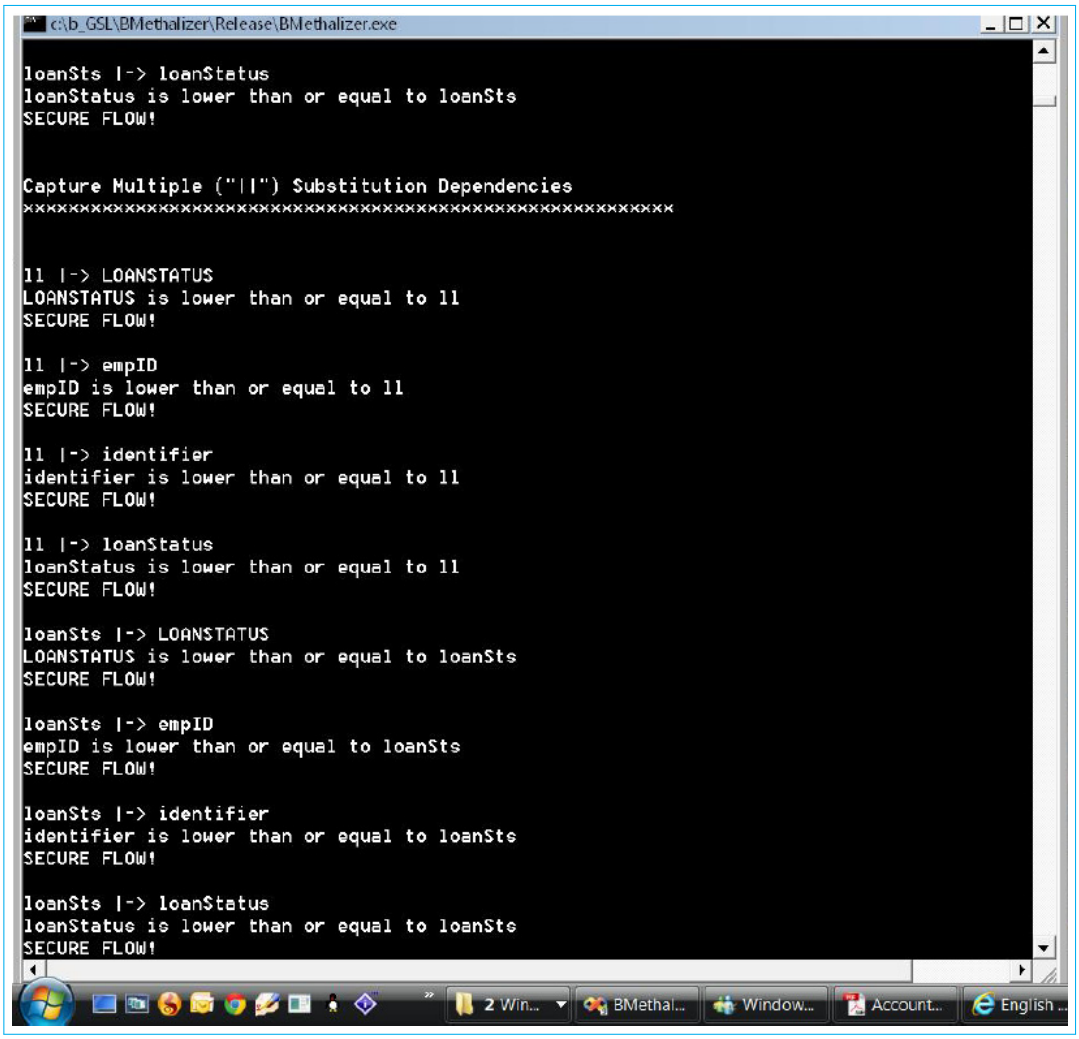


Figure 16: *Loan_CredAccounts : Flow Analysis*

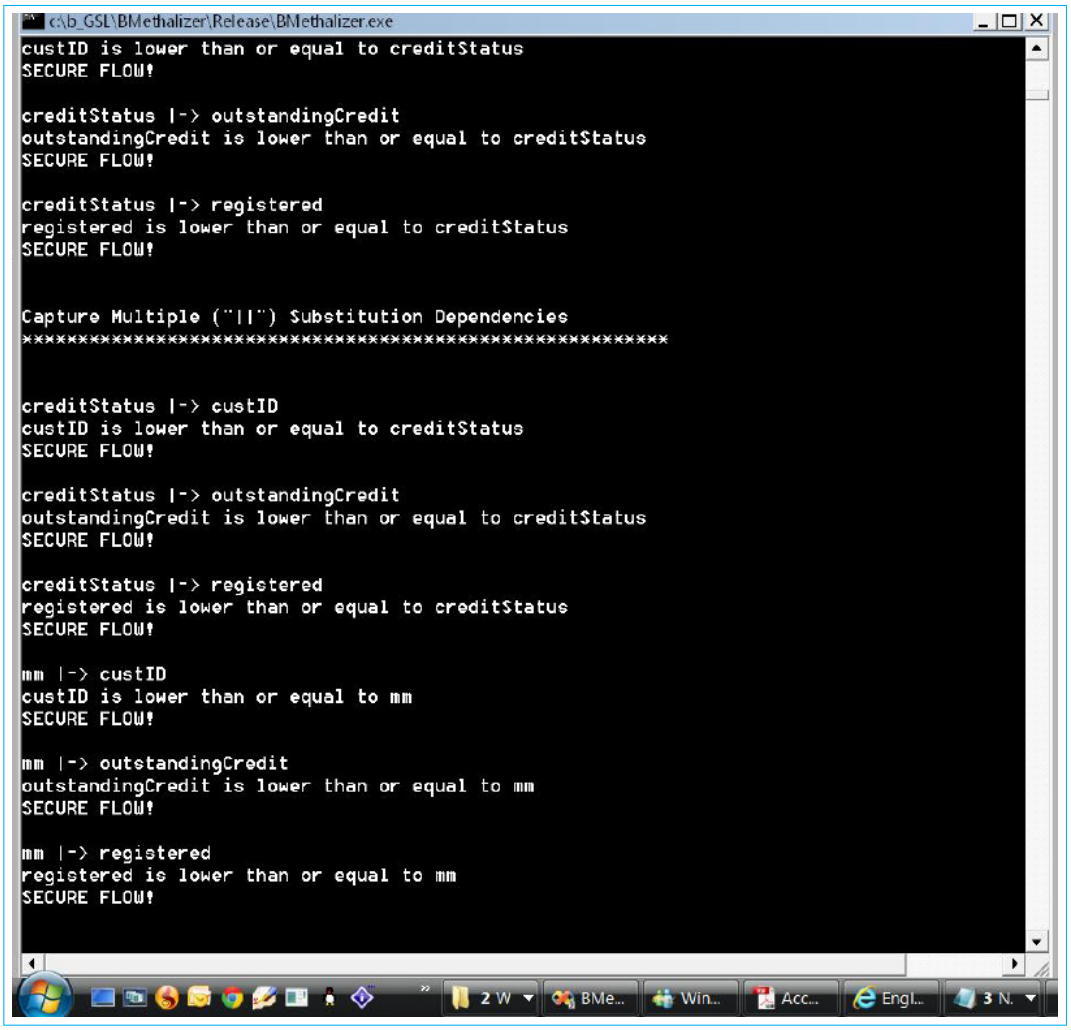


Figure 17: MACHINE *Loan_CredAccounts* : *Flow Analysis*

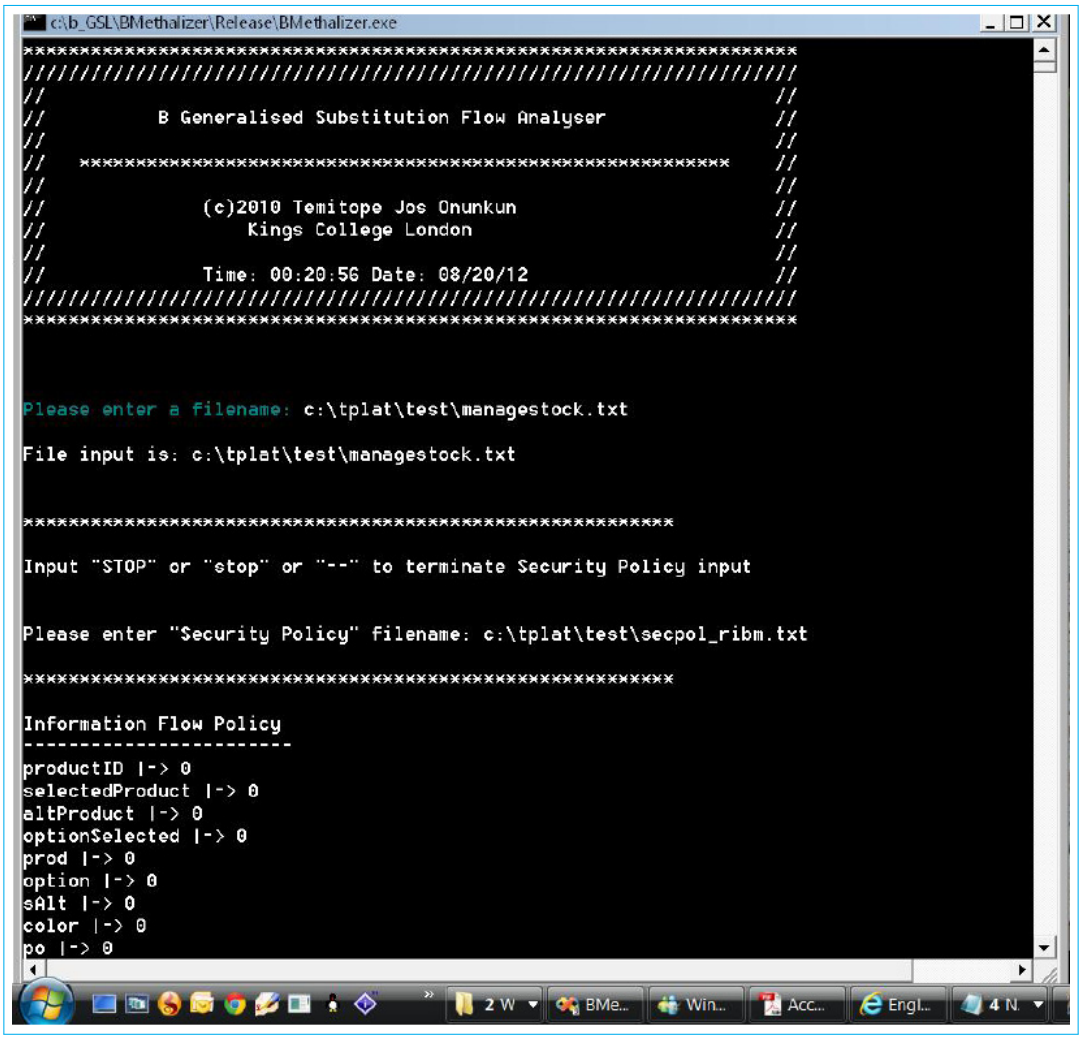


Figure 18: *ManageStock* : *Input Files*

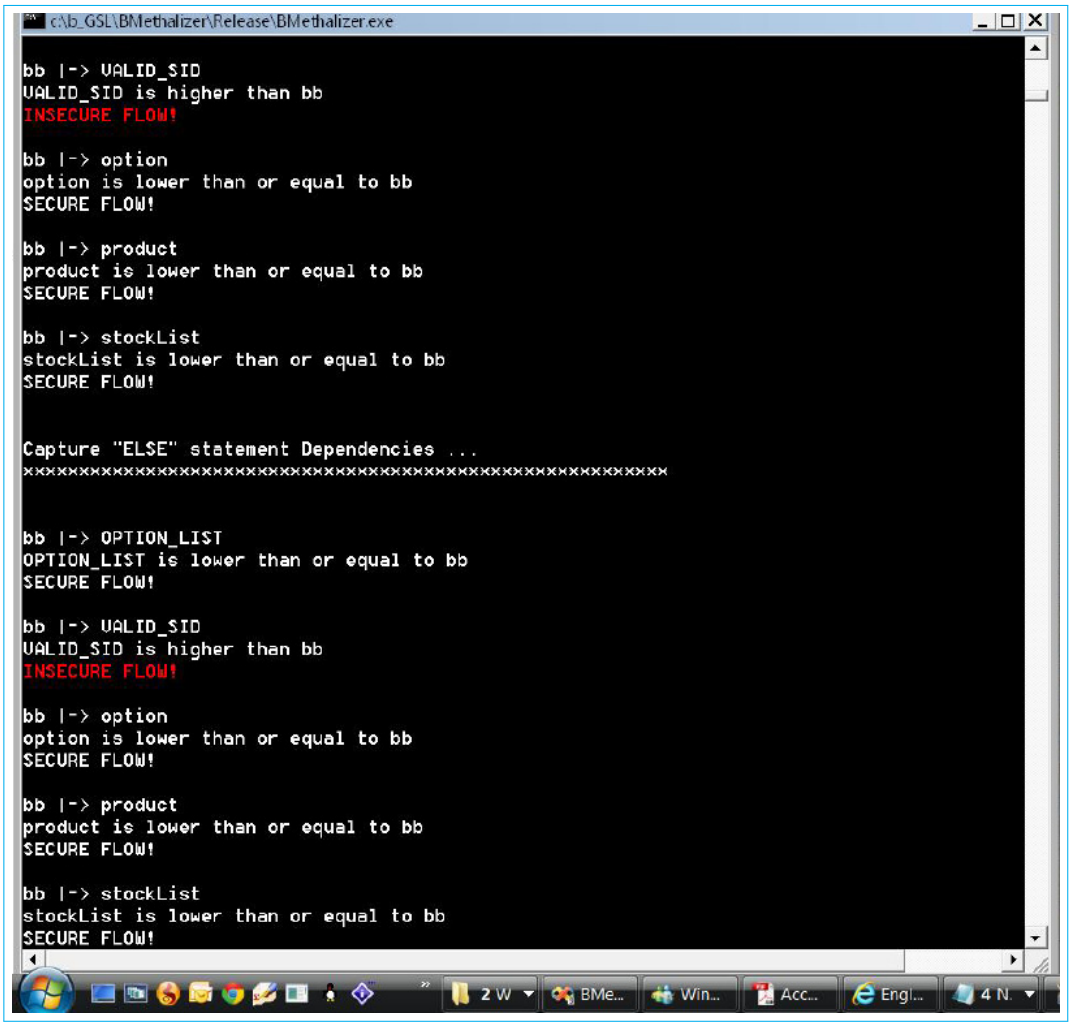


Figure 19: *ManageStock* : *Flow Analysis*

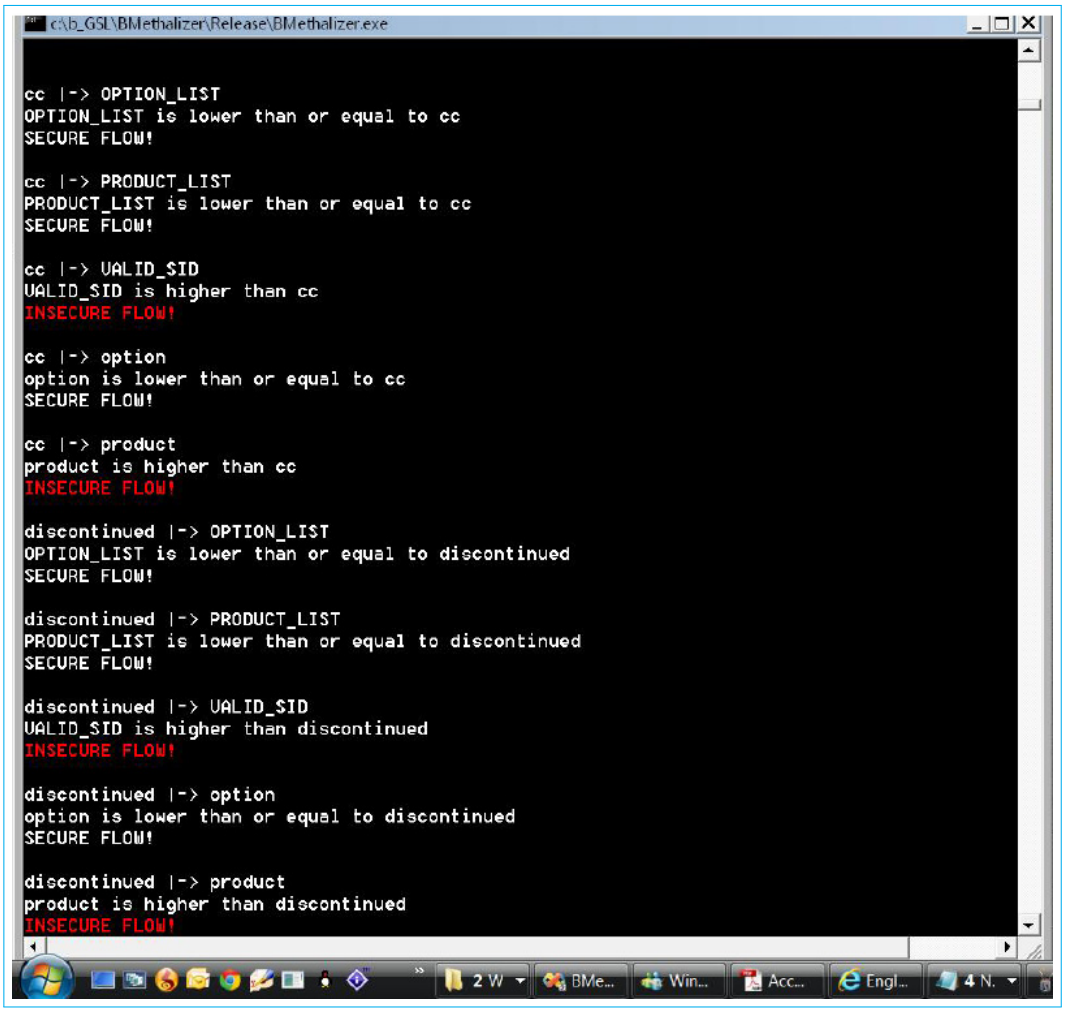


Figure 20: *ManageStock : Flow Analysis*

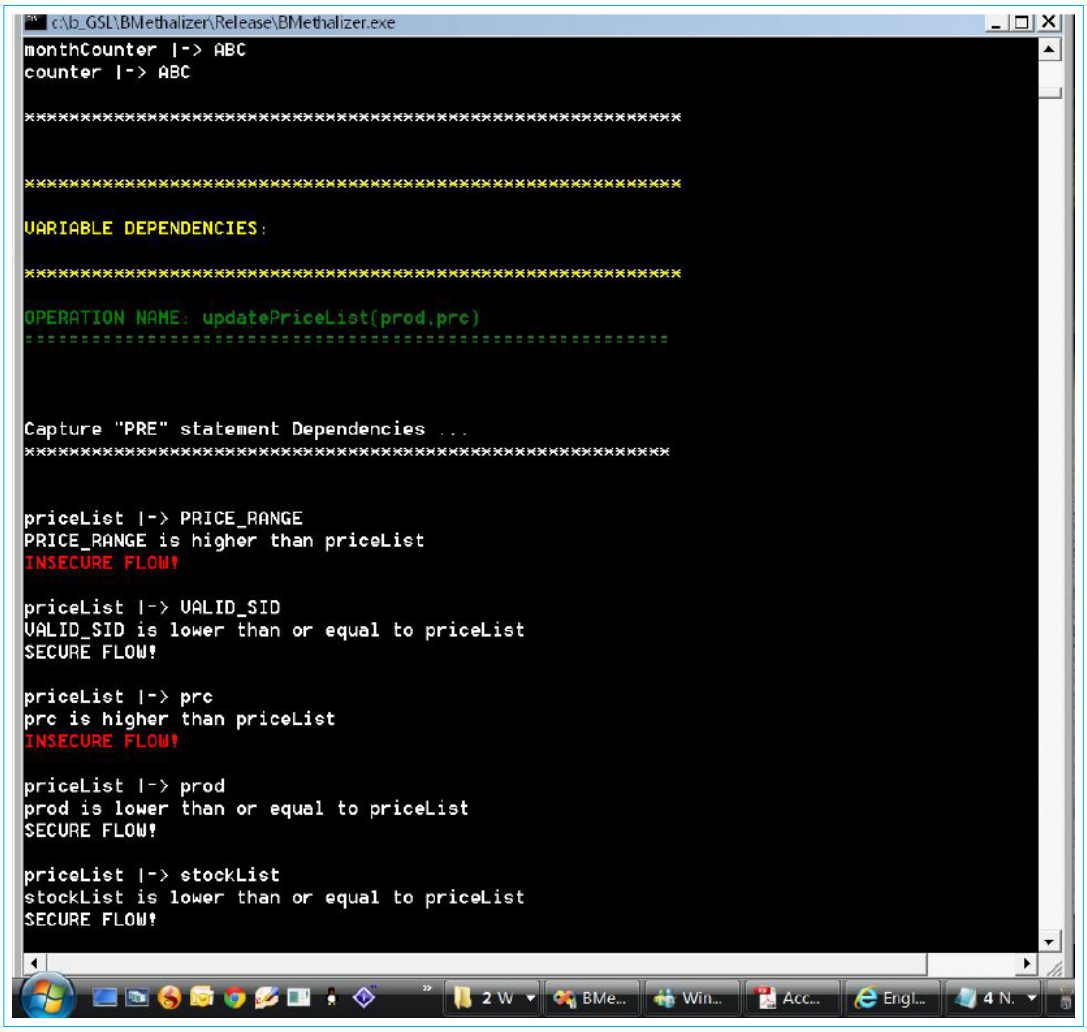


Figure 21: *ManageStock* : *Flow Analysis*

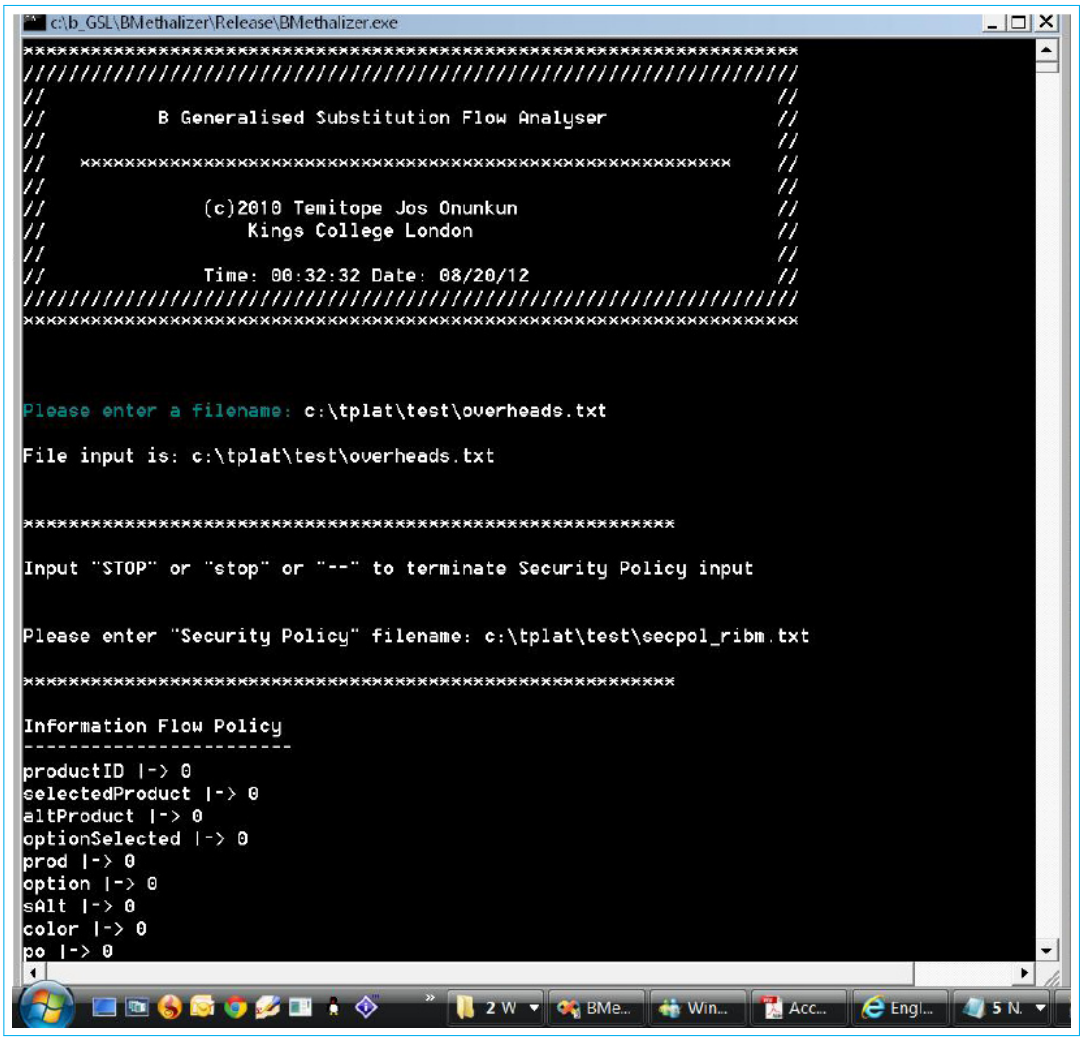


Figure 22: *Overheads : Input Files*

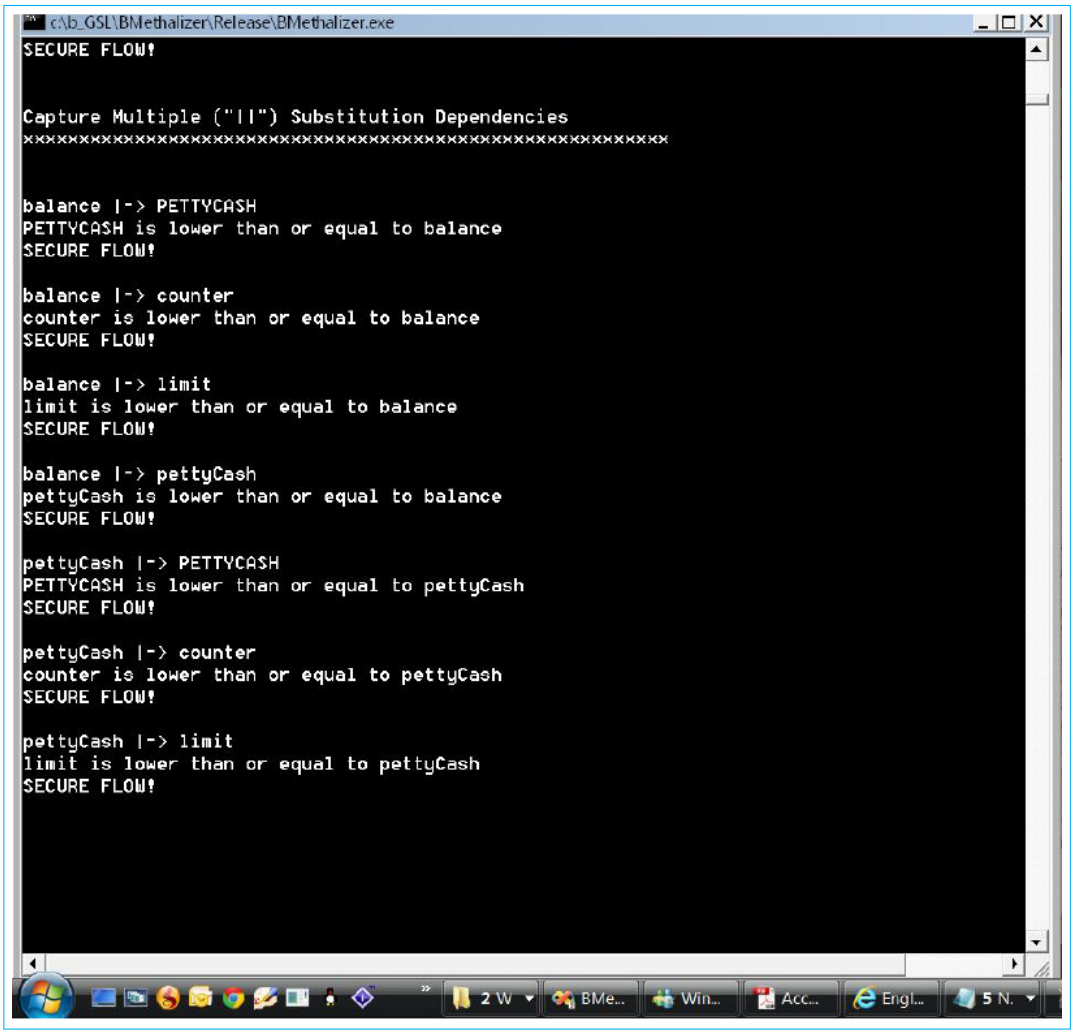


Figure 23: *Overheads : Flow Analysis*

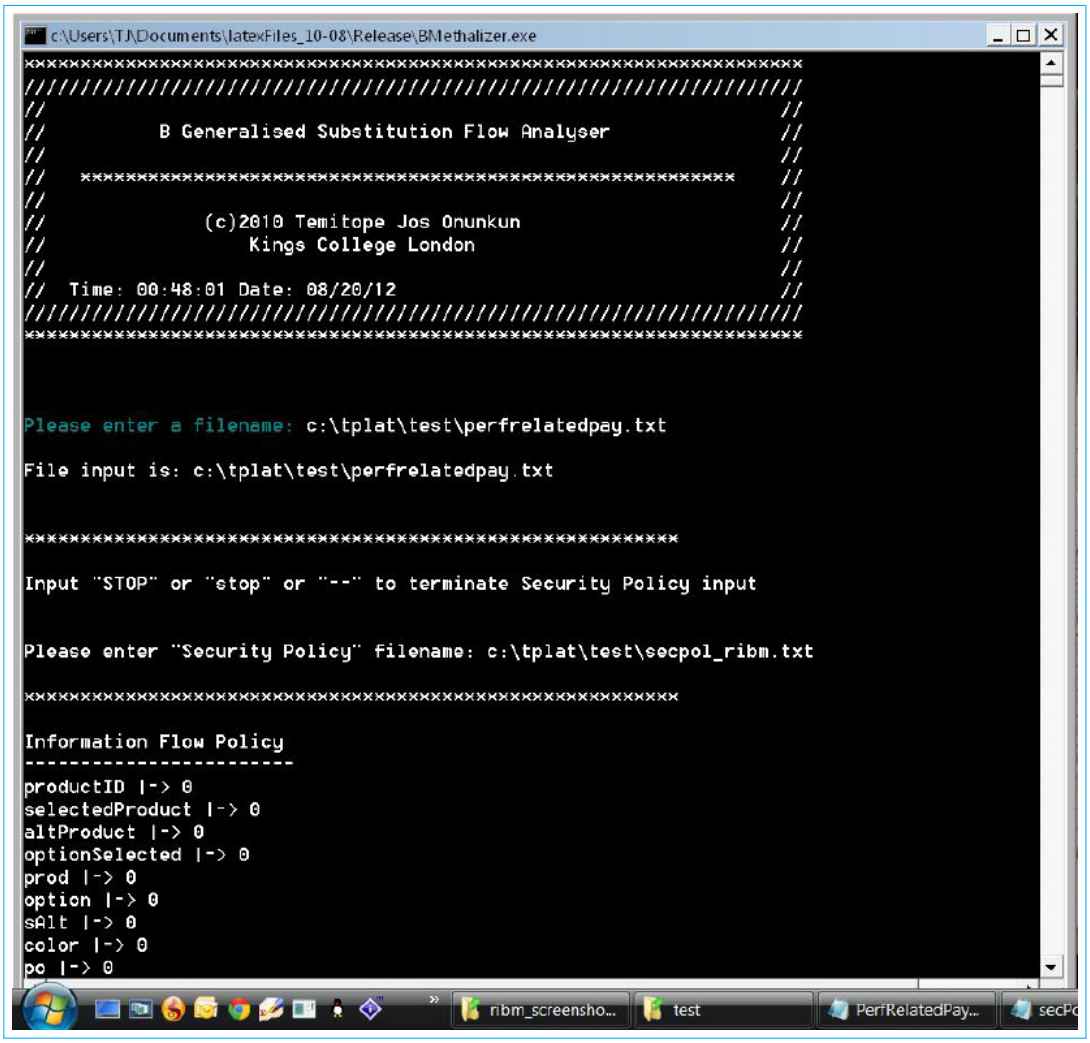


Figure 24: $PerfRelatedPay: \ Input \ Files$

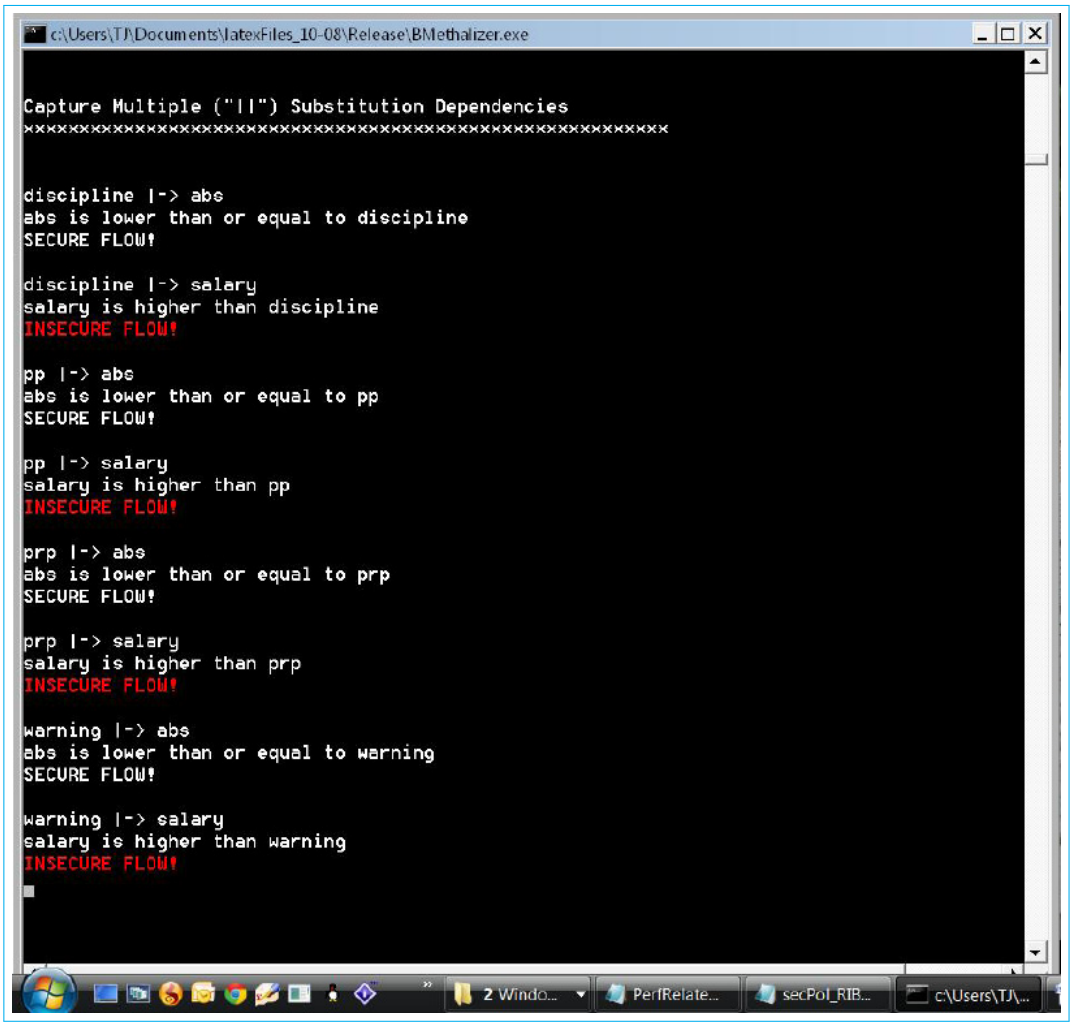


Figure 25: $PerfRelatedPay: \ Flow \ Analysis$

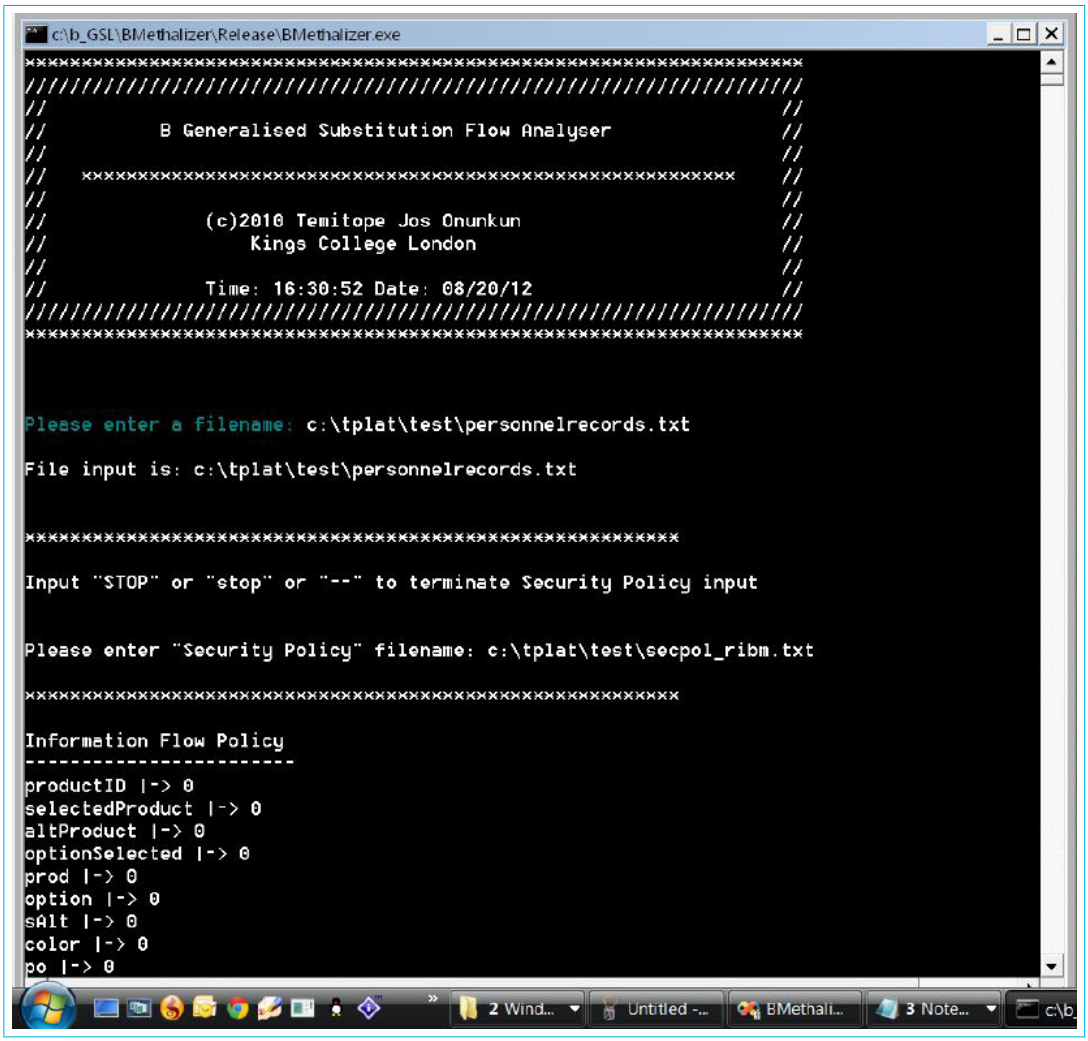


Figure 26: *PersonnelRecords : Input Files*

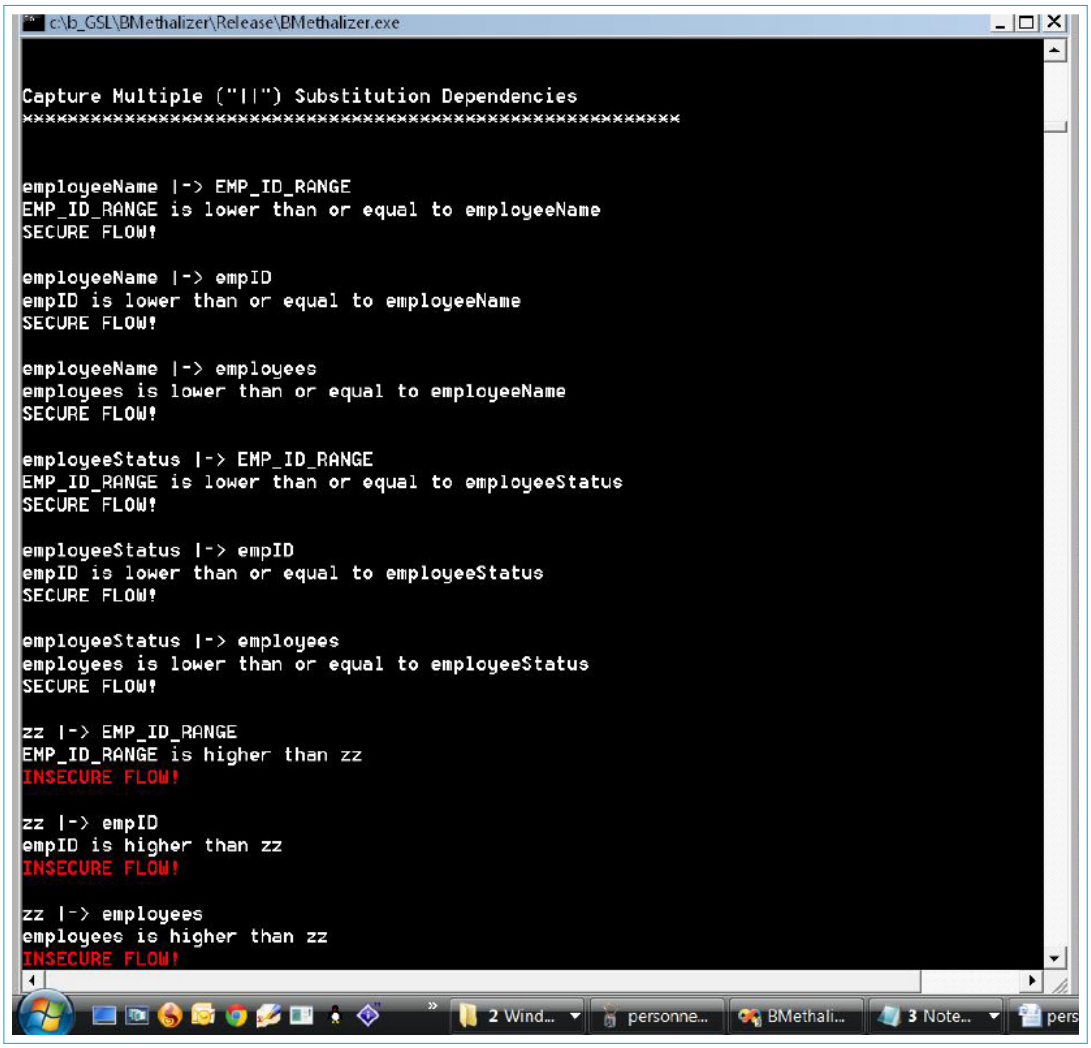


Figure 27: *PersonnelRecords* : *Flow Analysis*

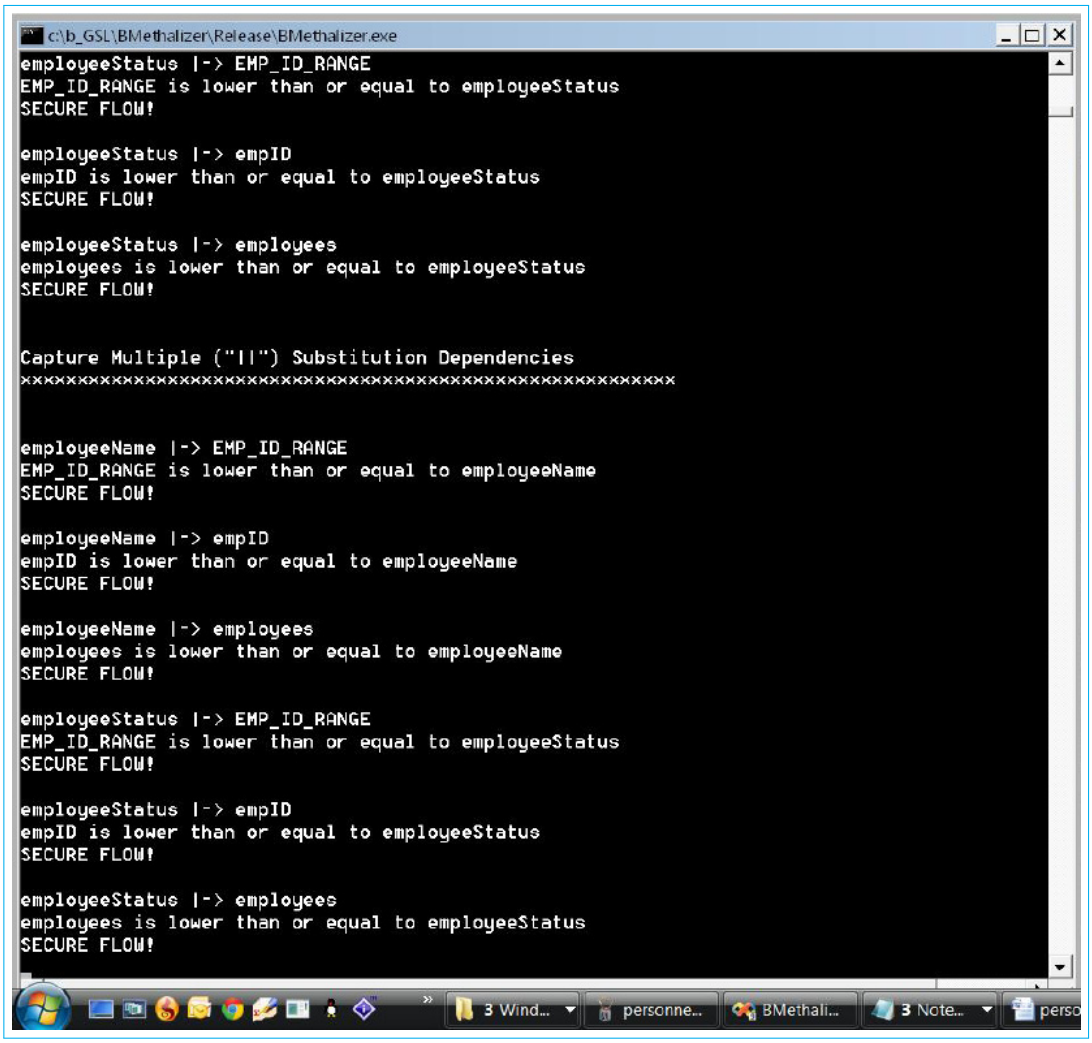


Figure 28: *PersonnelRecords* : *Flow Analysis*

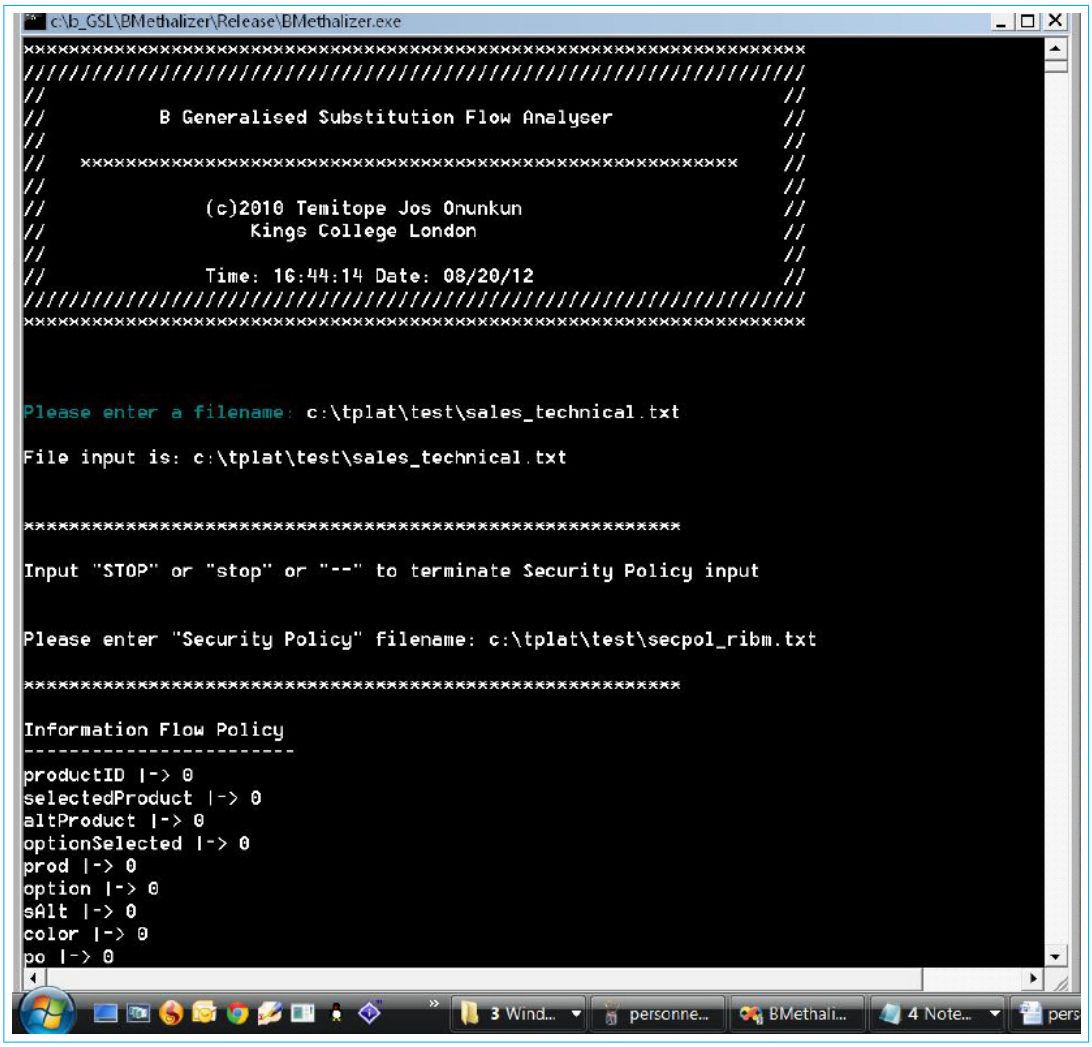


Figure 29: *Sales_Technical : Input Files*

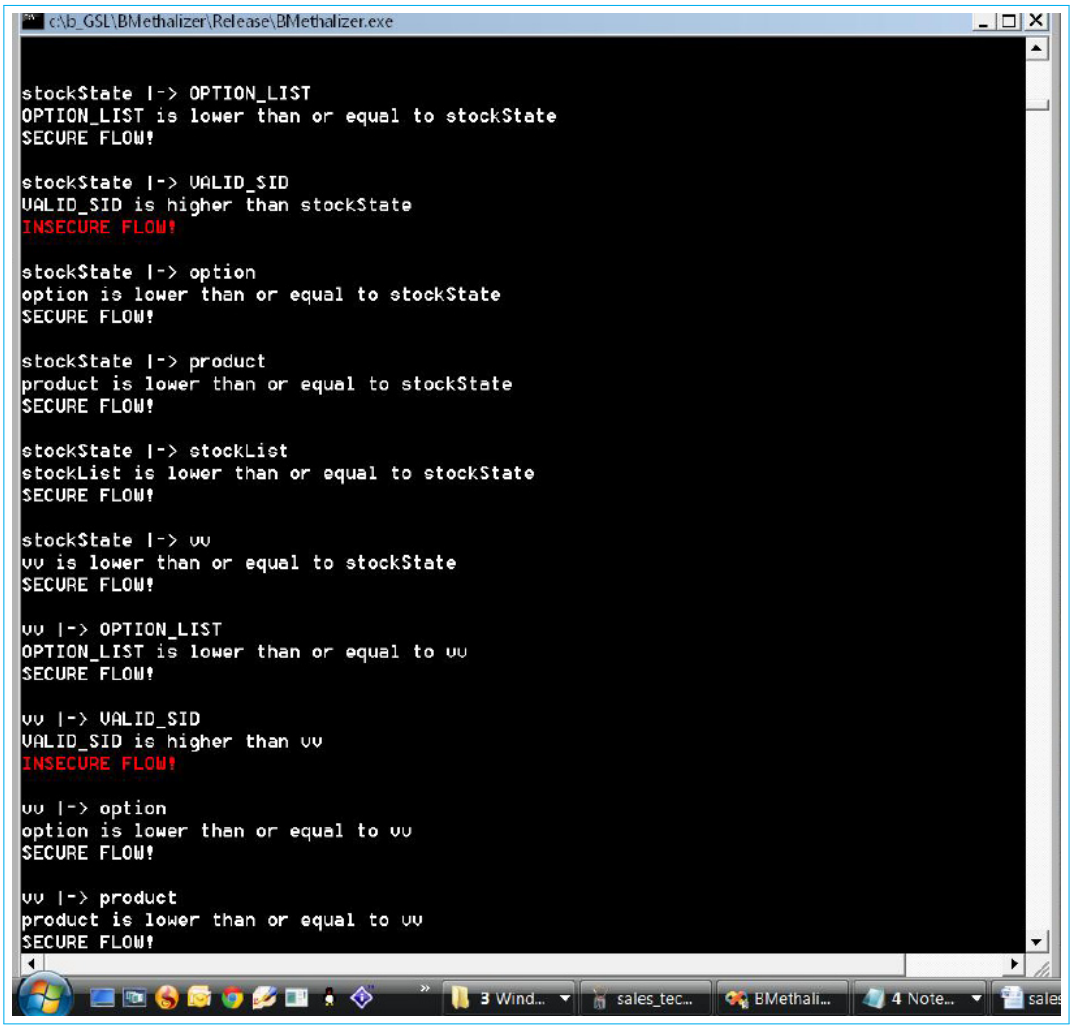


Figure 30: *Sales_Technical : Flow Analysis*

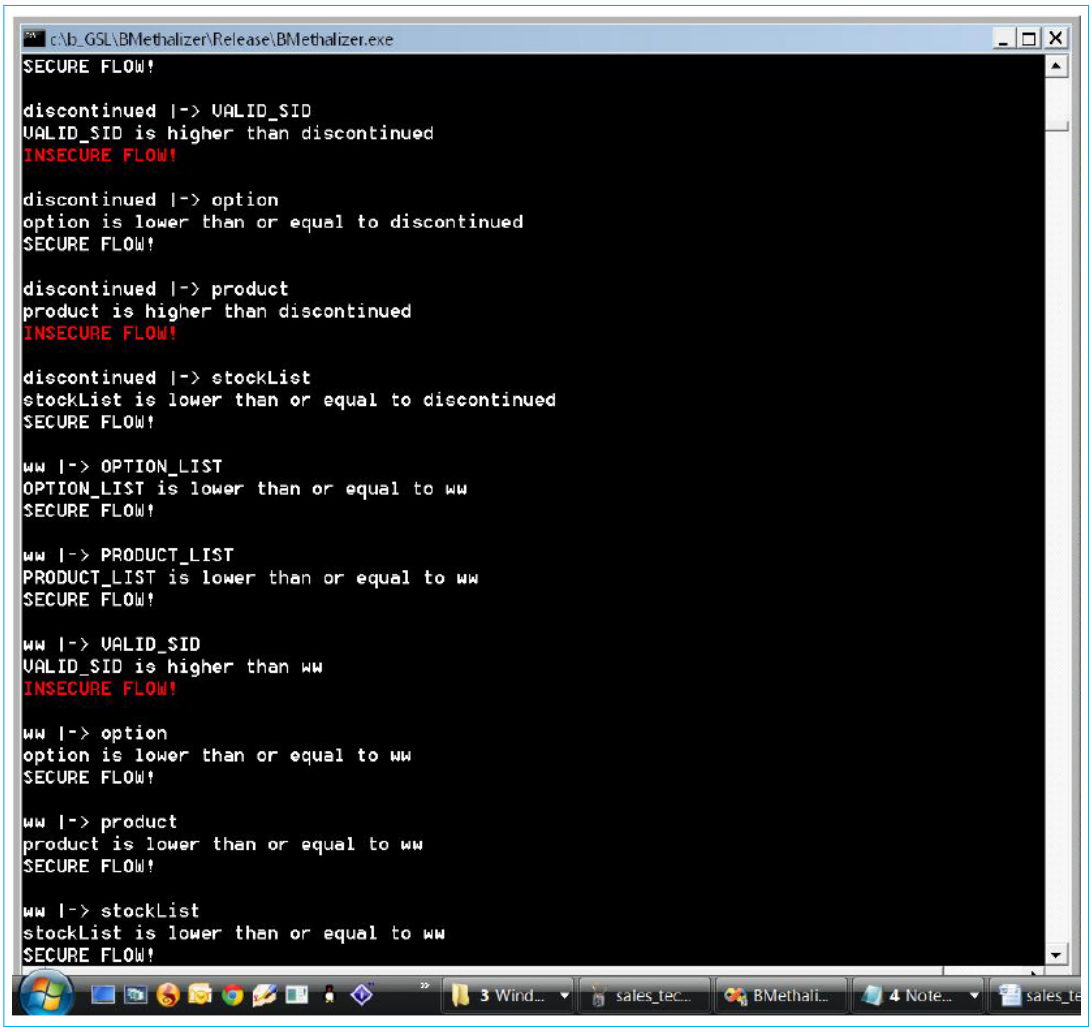


Figure 31: *Sales_Technical : Flow Analysis*

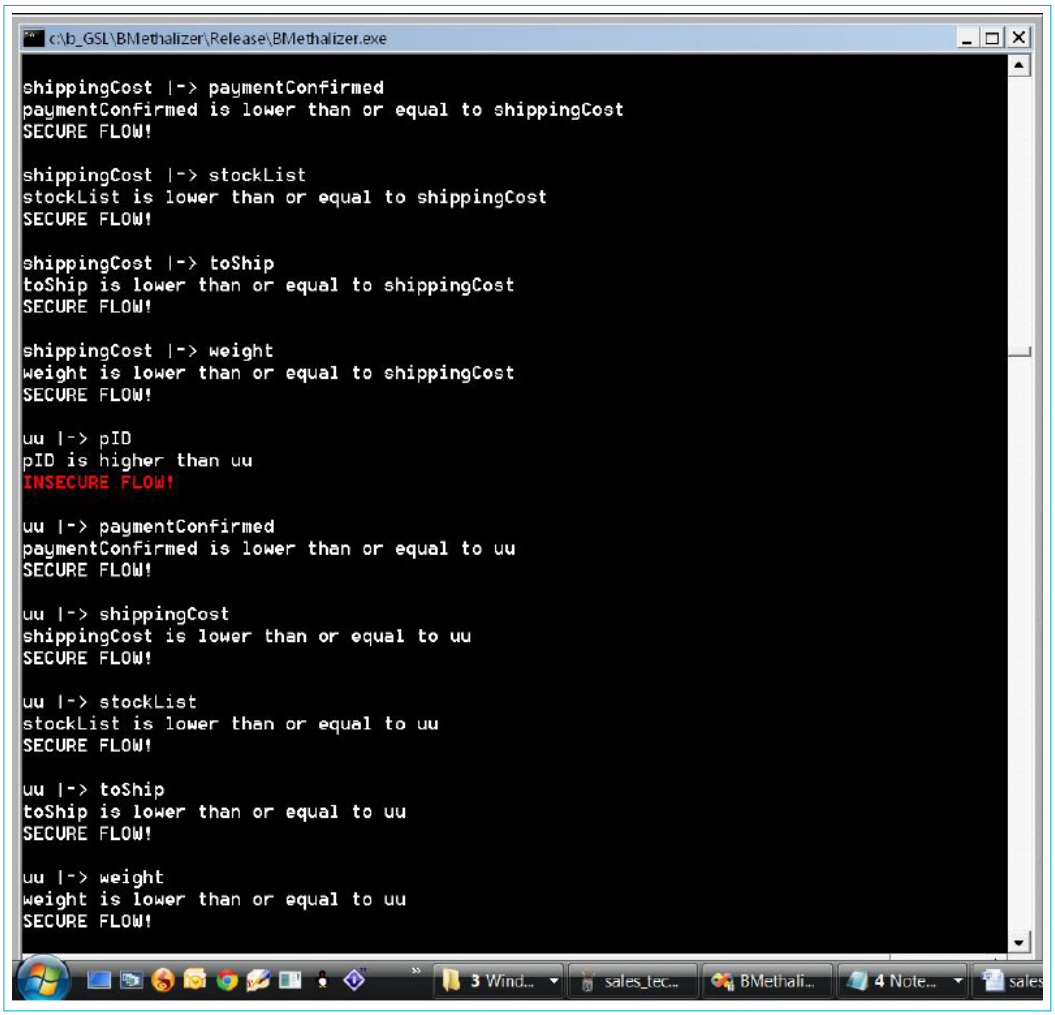


Figure 32: *Sales_Technical : Flow Analysis*

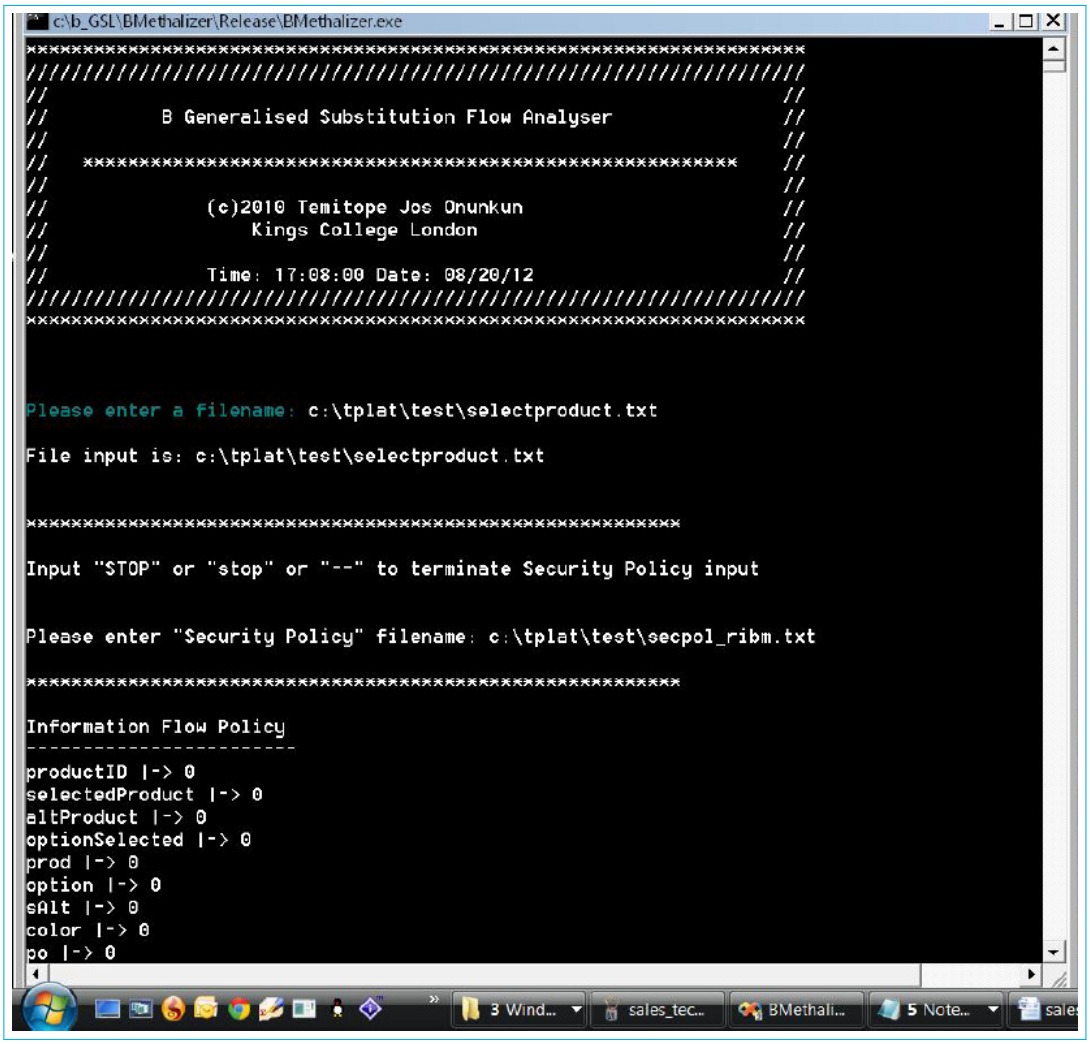


Figure 33: *SelectProduct : Input Files*

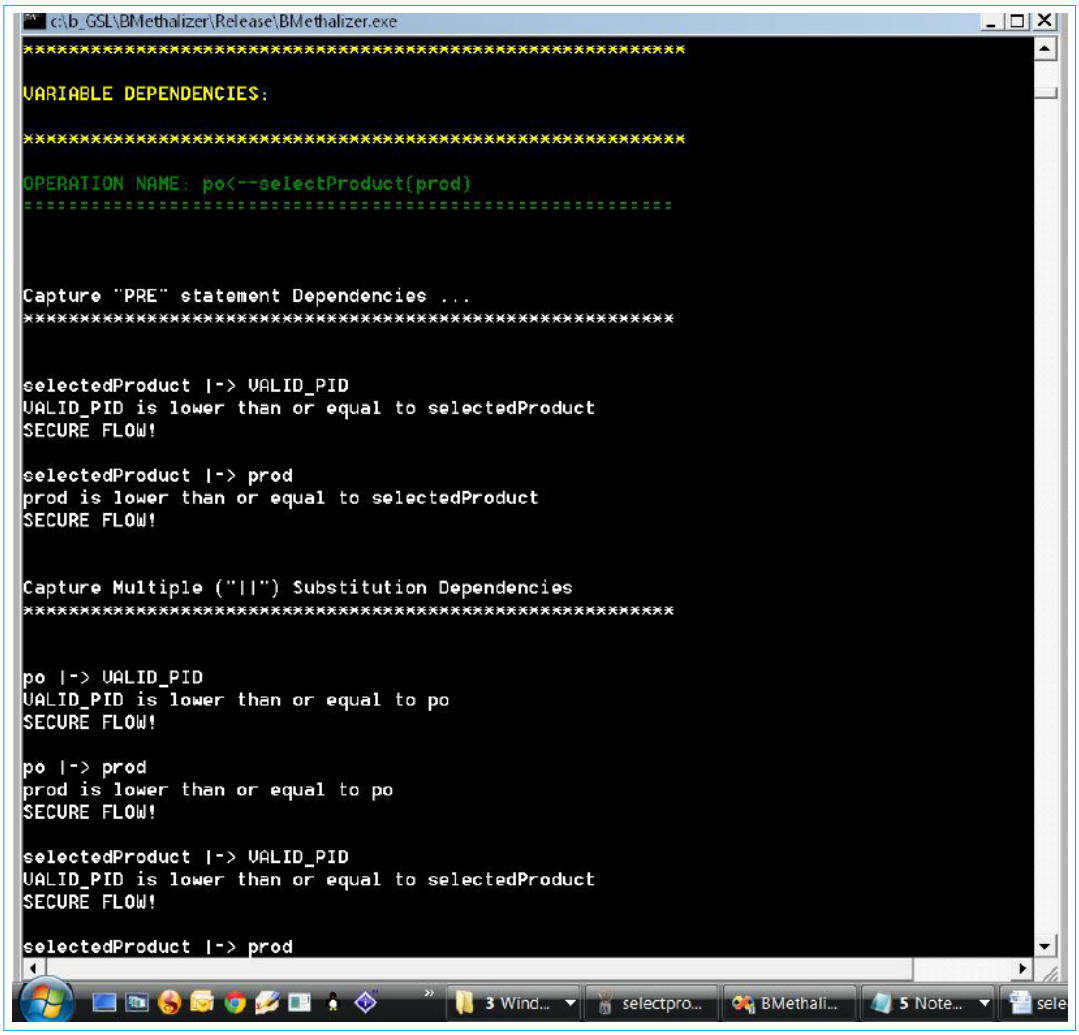


Figure 34: *SelectProduct* : *Flow Analysis*

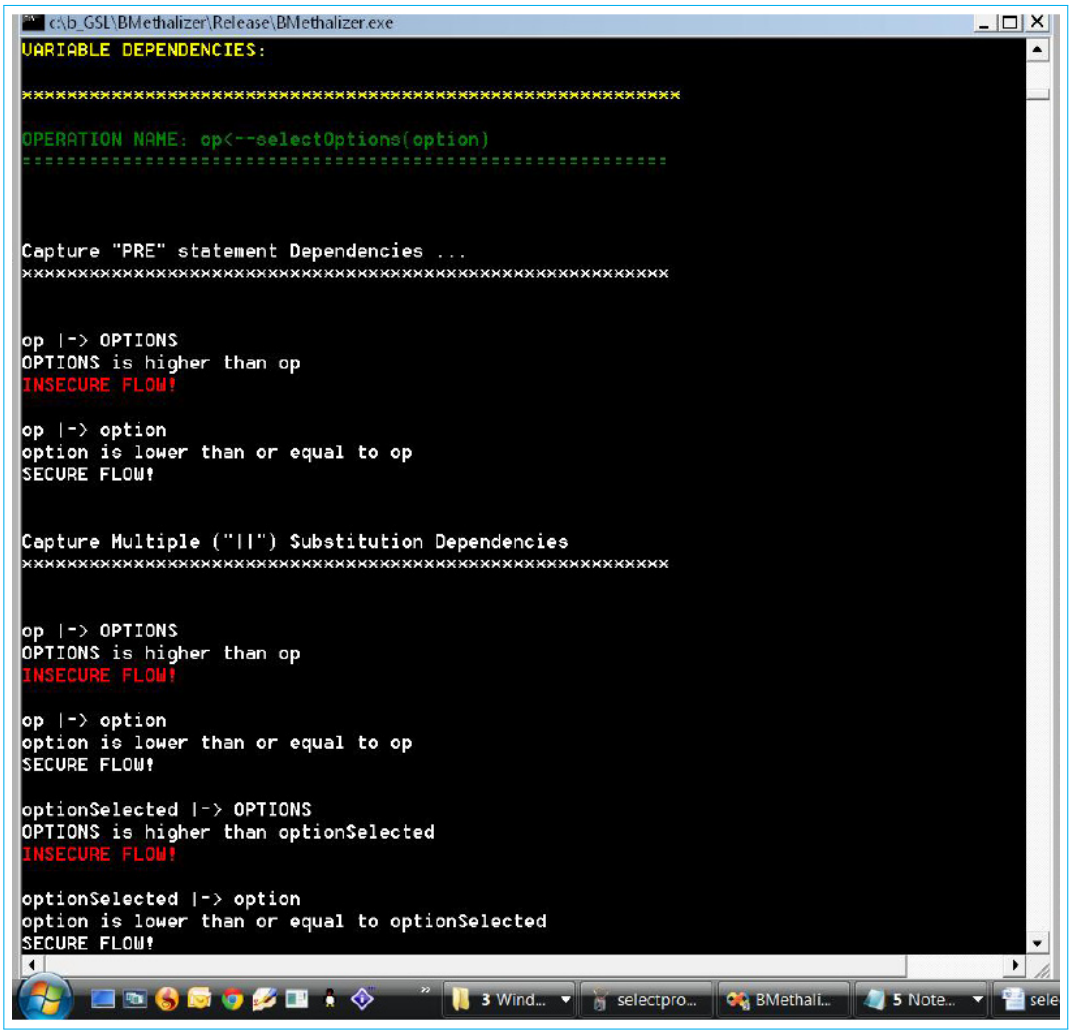


Figure 35: *SelectProduct* : *Flow Analysis*

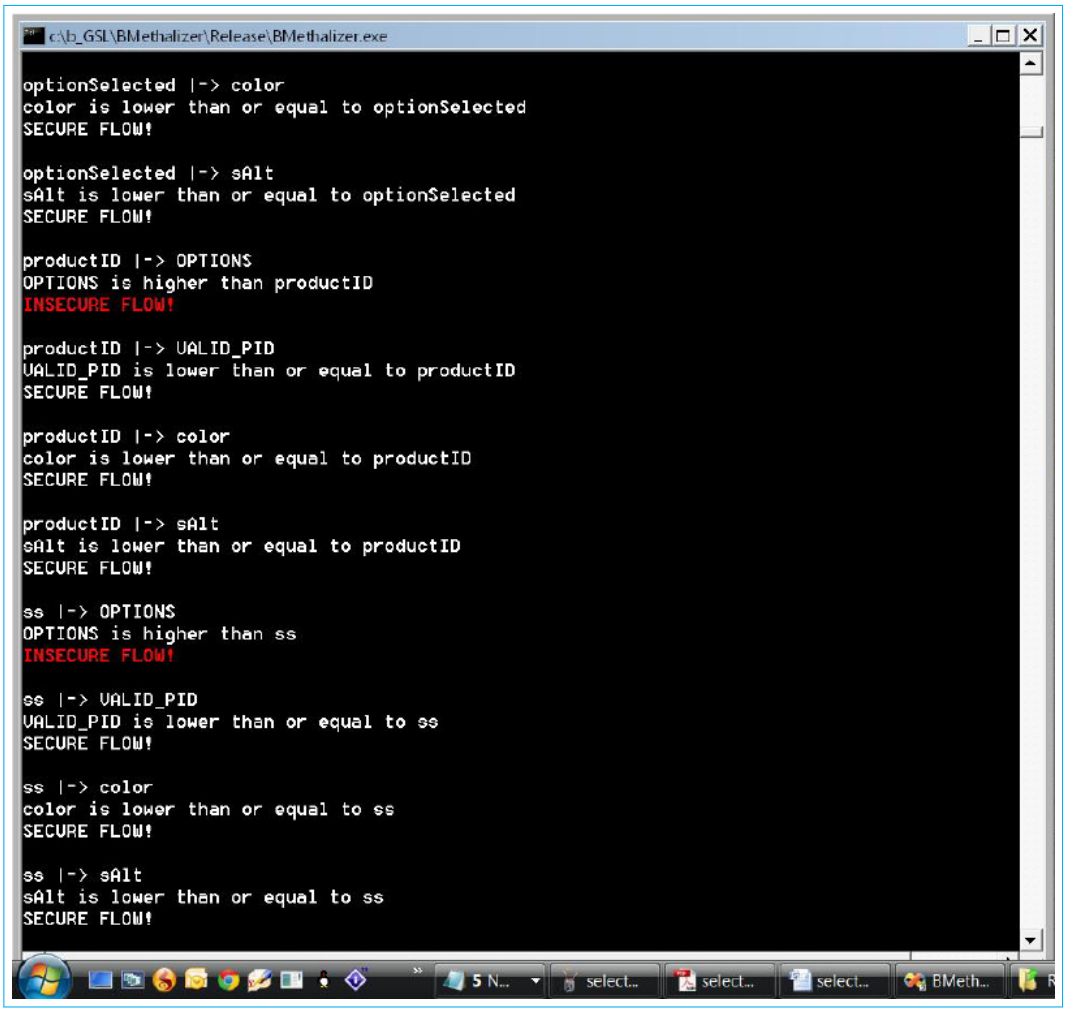


Figure 36: *SelectProduct : Flow Analysis*

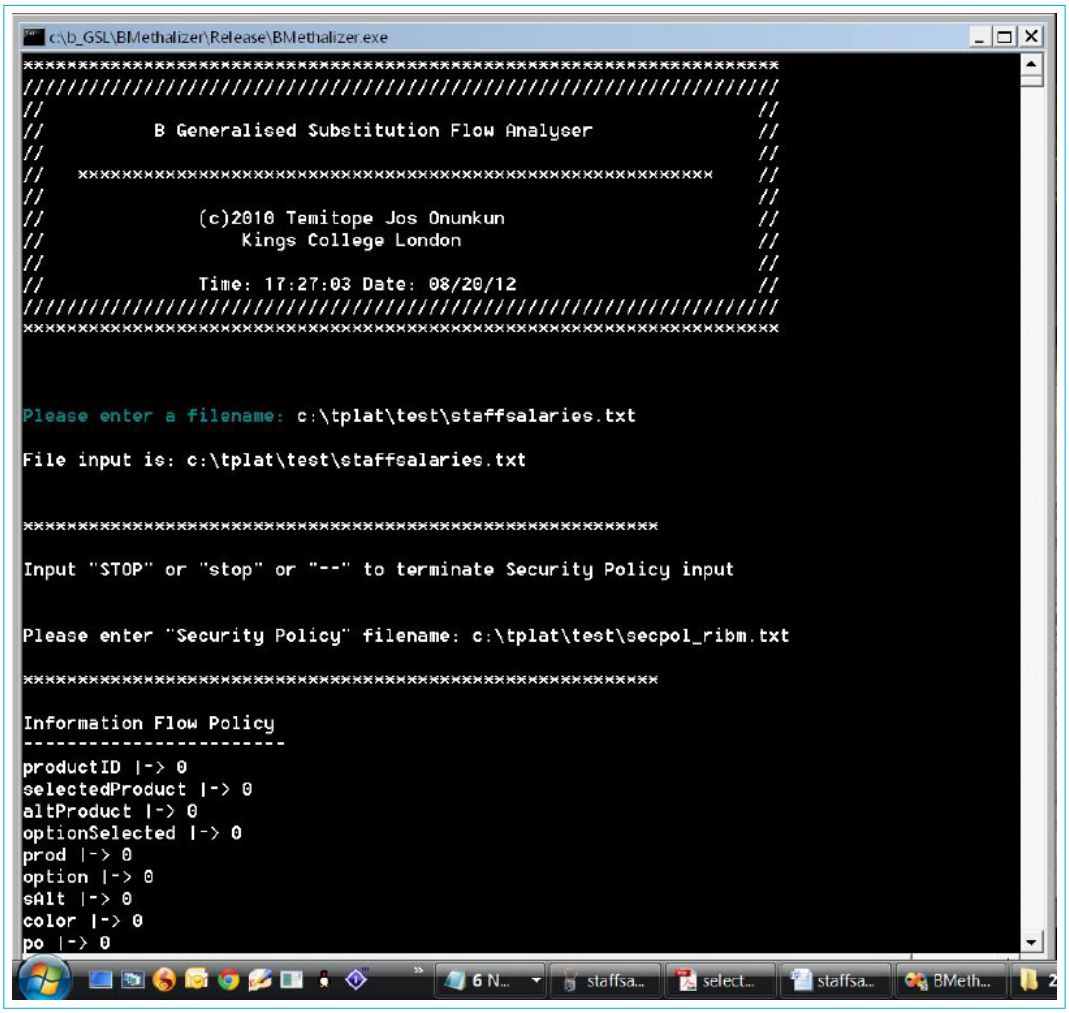


Figure 37: *StaffSalaries : Input Files*

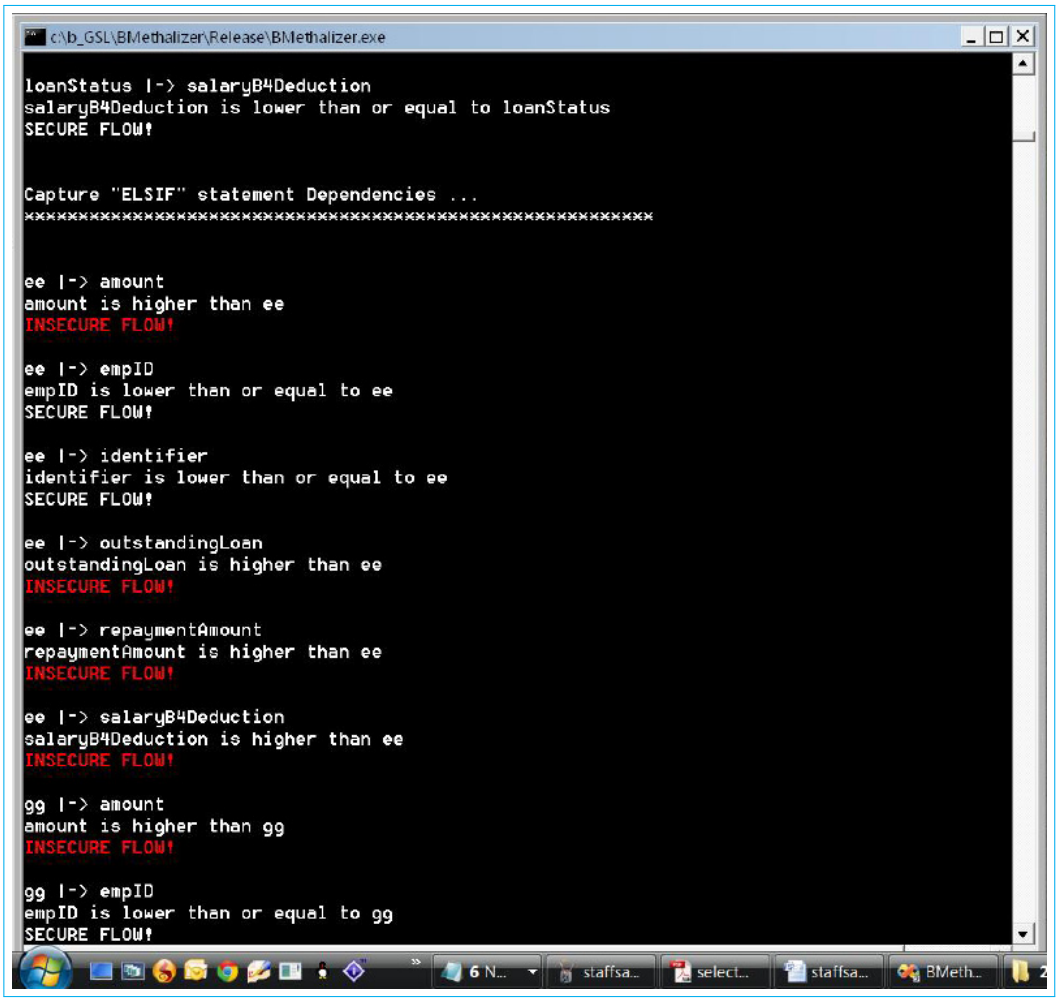


Figure 38: $Staff Salaries: Flow\ Analysis$

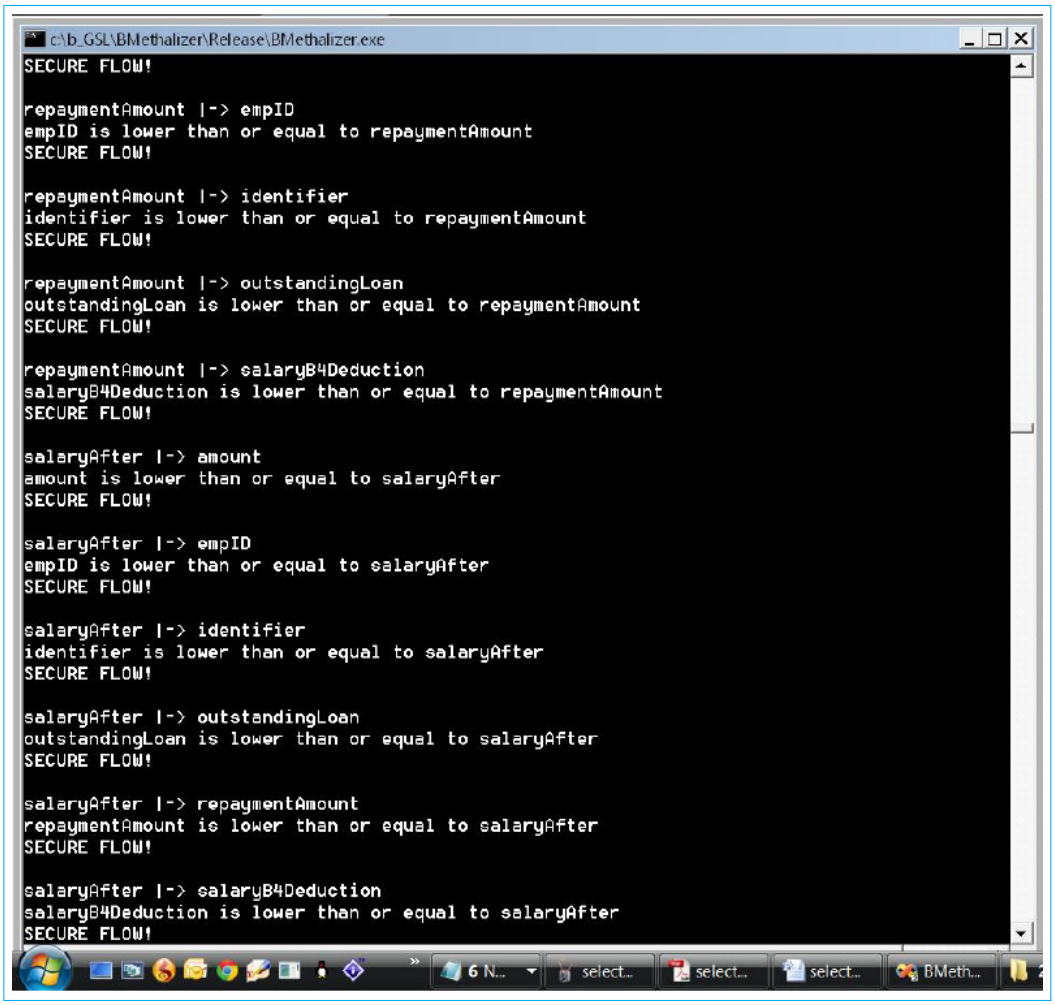


Figure 39: $Staff Salaries: \ Flow \ Analysis$

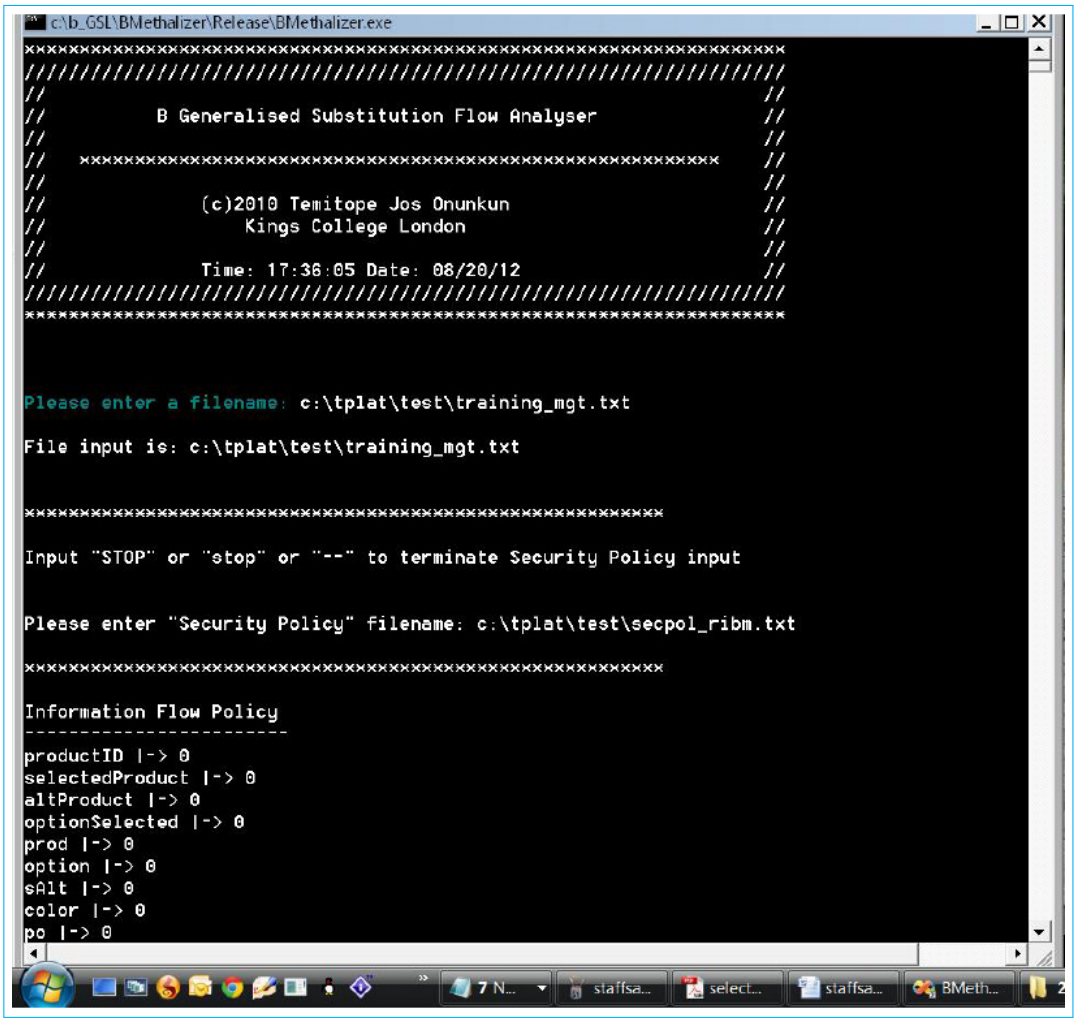


Figure 40: *Training_Mgt : Input Files*

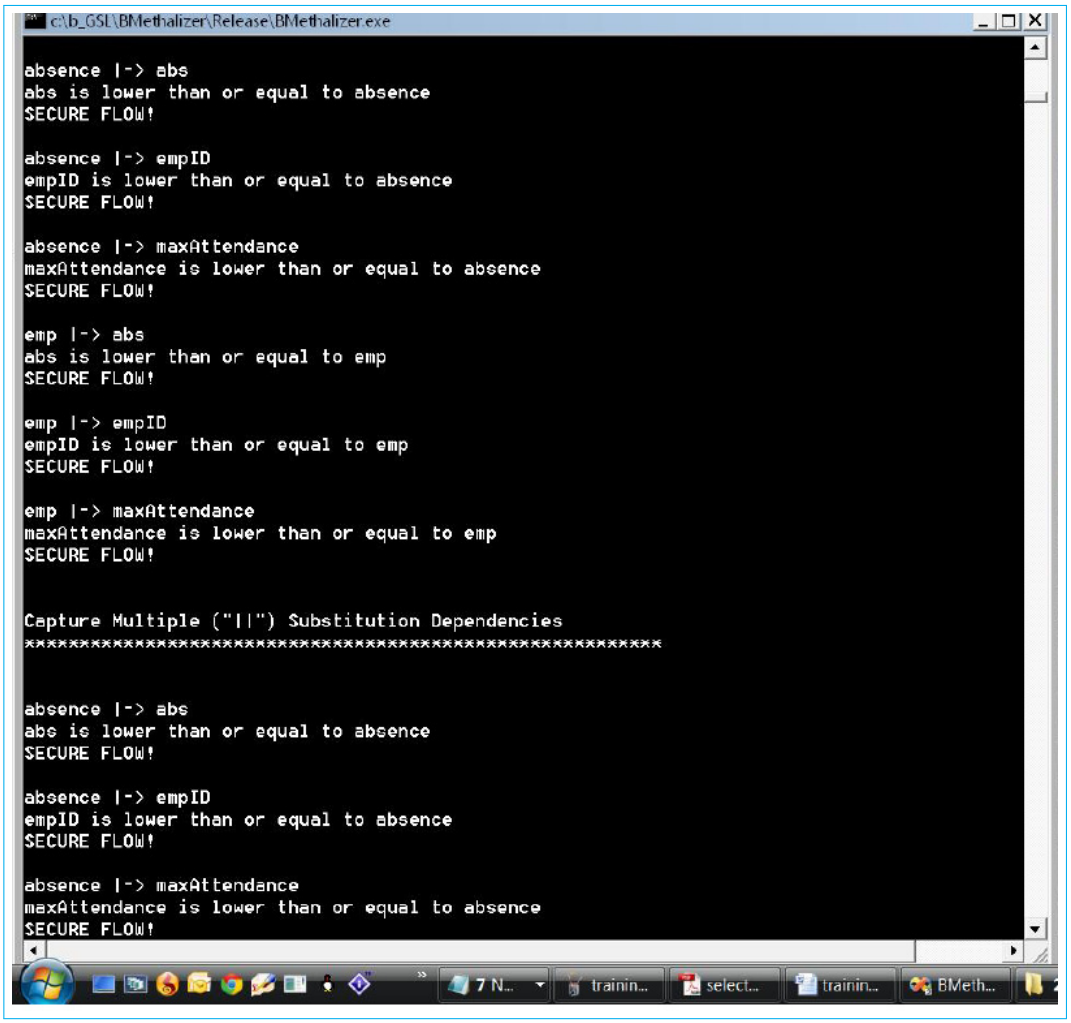


Figure 41: *Training_Mgt* : *Flow Analysis*

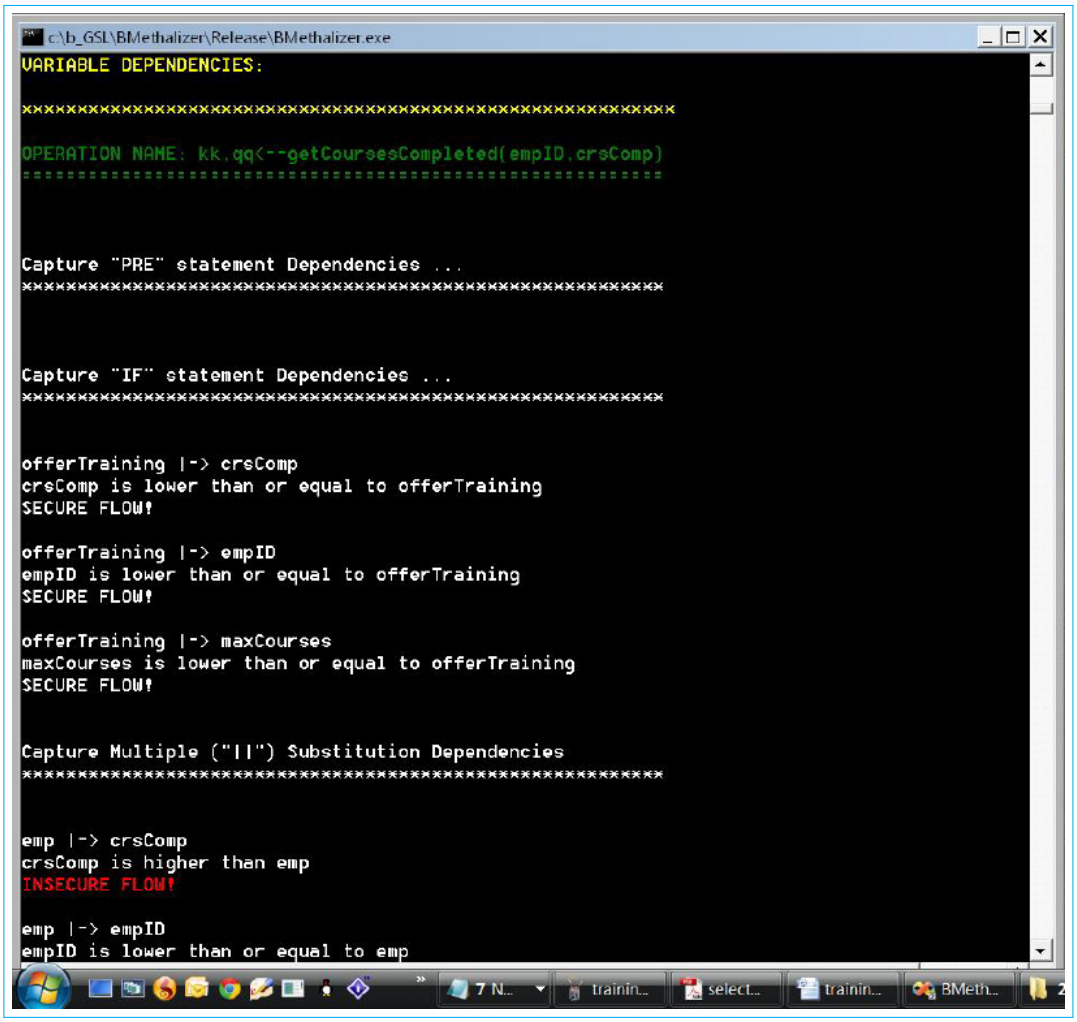


Figure 42: *Training_Mgt* : *Flow Analysis*

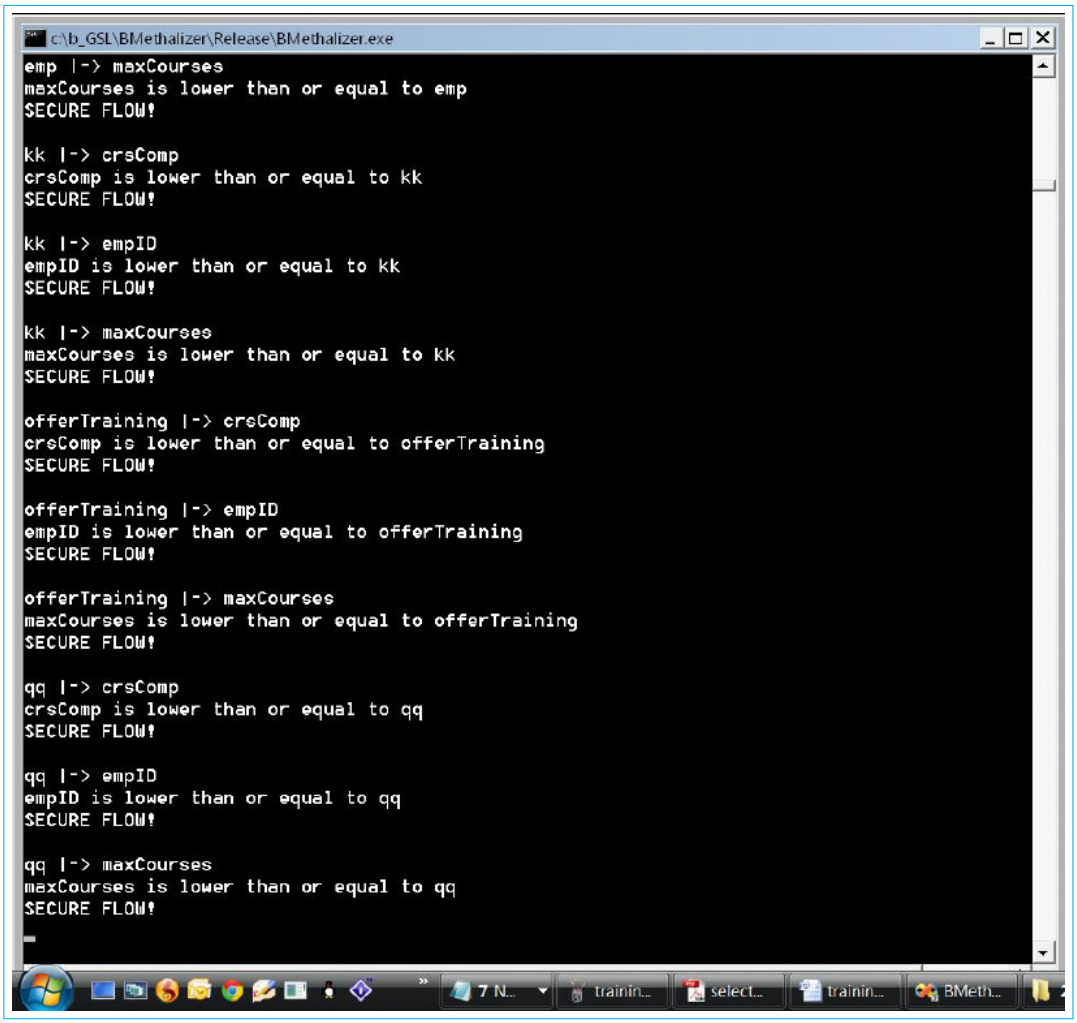


Figure 43: *Training_Mgt* : *Flow Analysis*

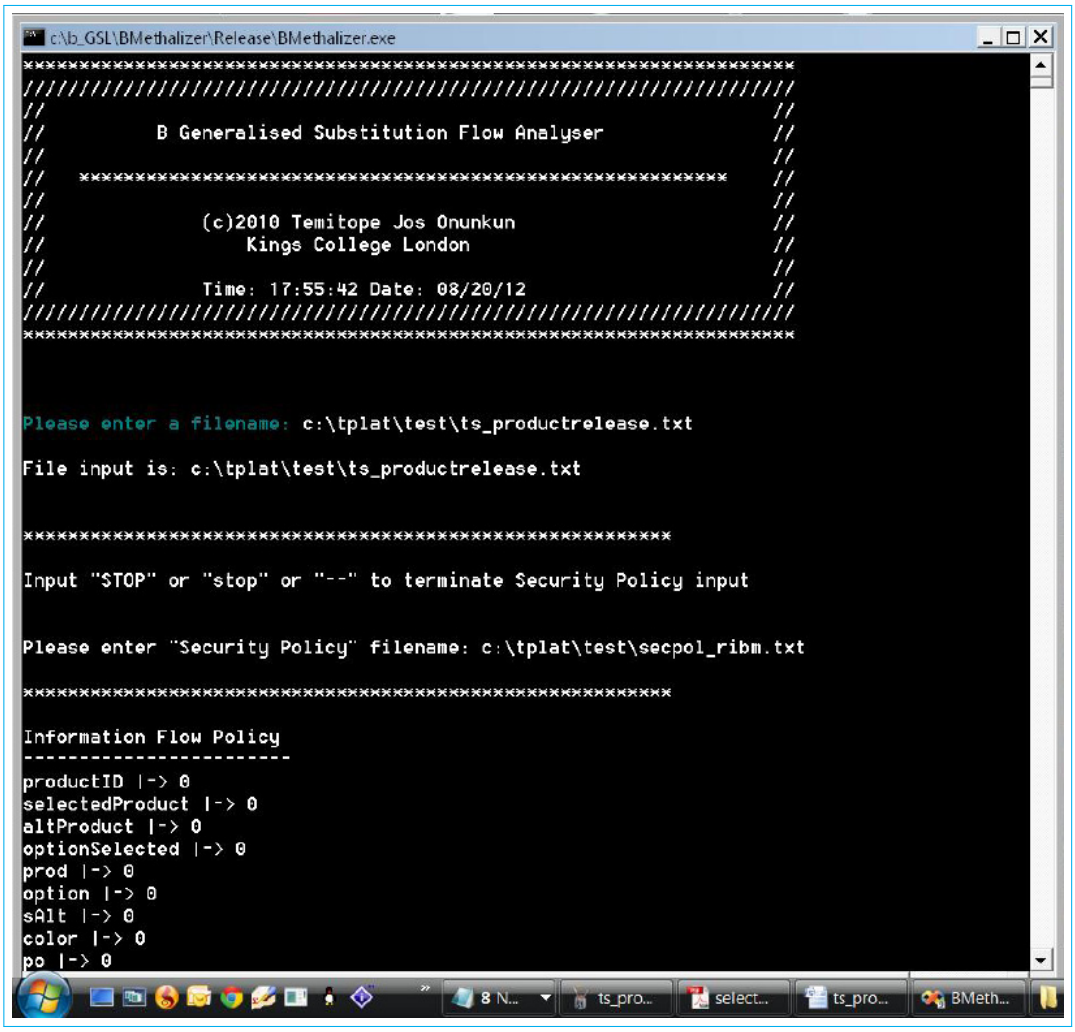


Figure 44: $Ts\_ProductRelease: Input\ Files$

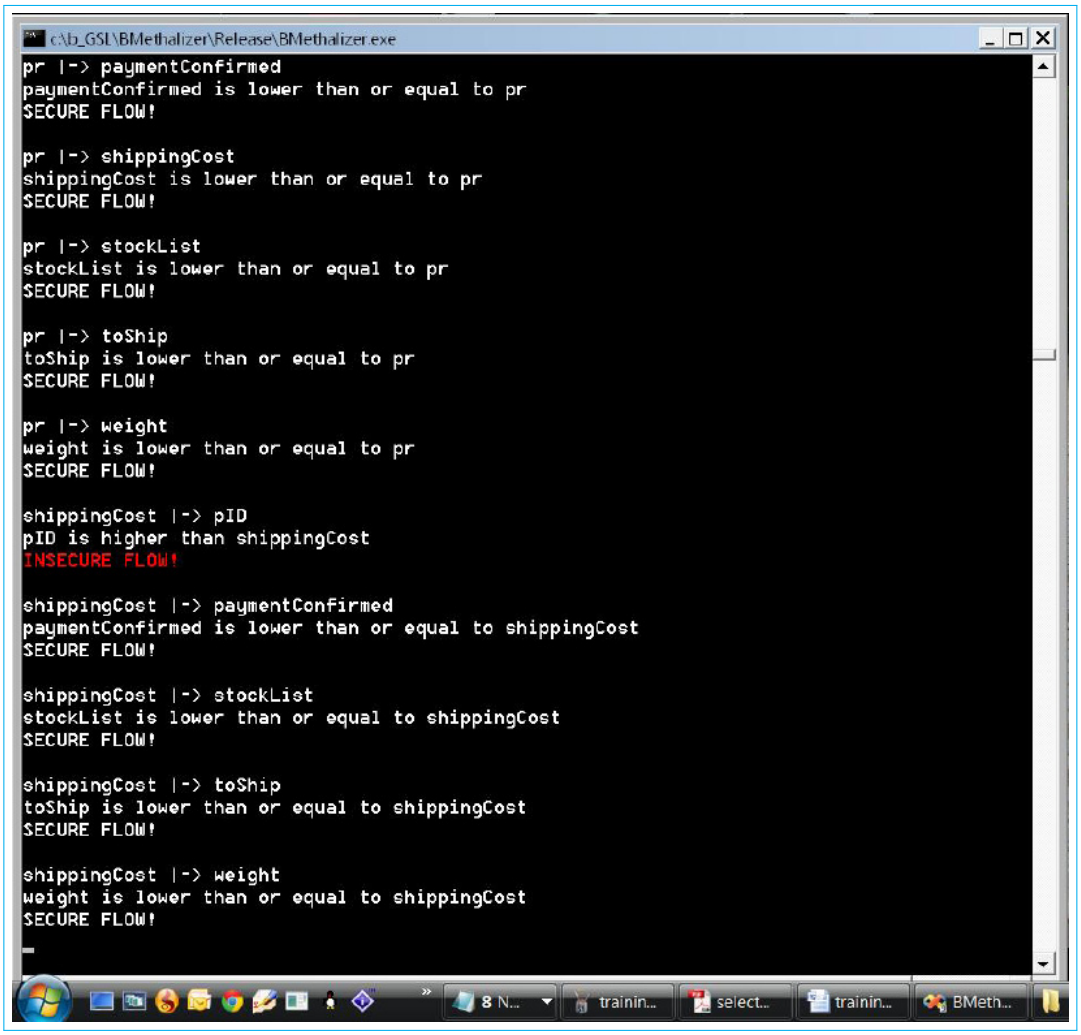


Figure 45: *Ts_ProductRelease : Flow Analysis*

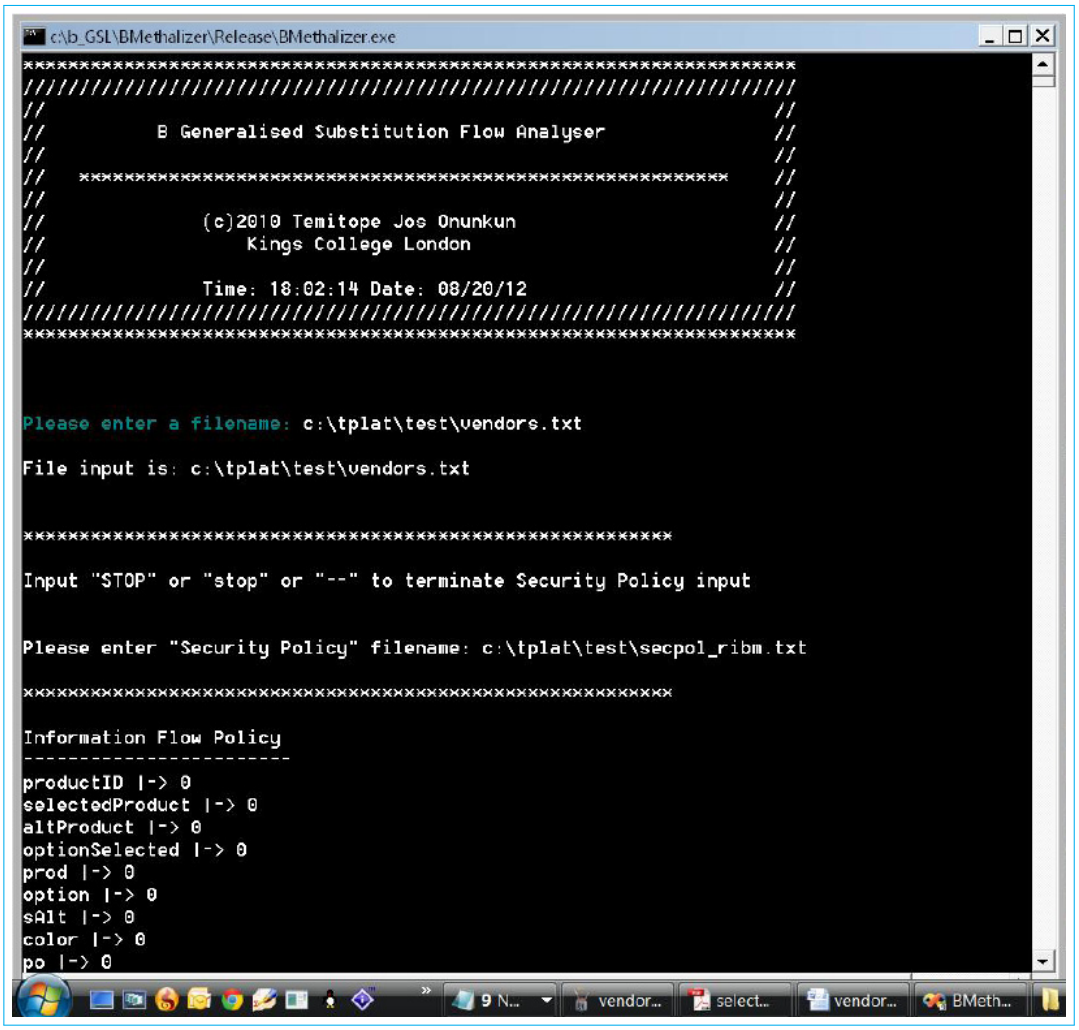


Figure 46: *Vendors* : *Input Files*

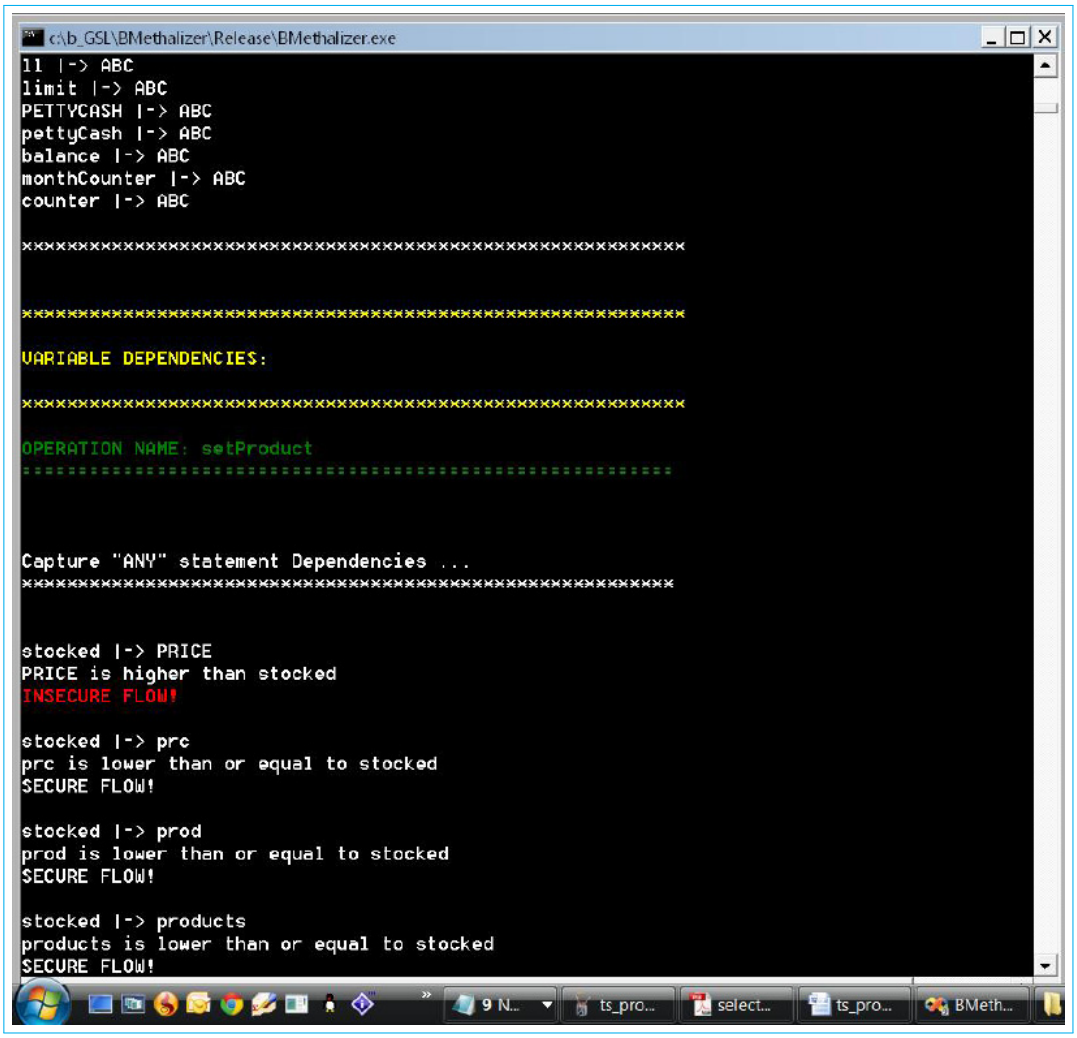


Figure 47: *Vendors : Flow Analysis*

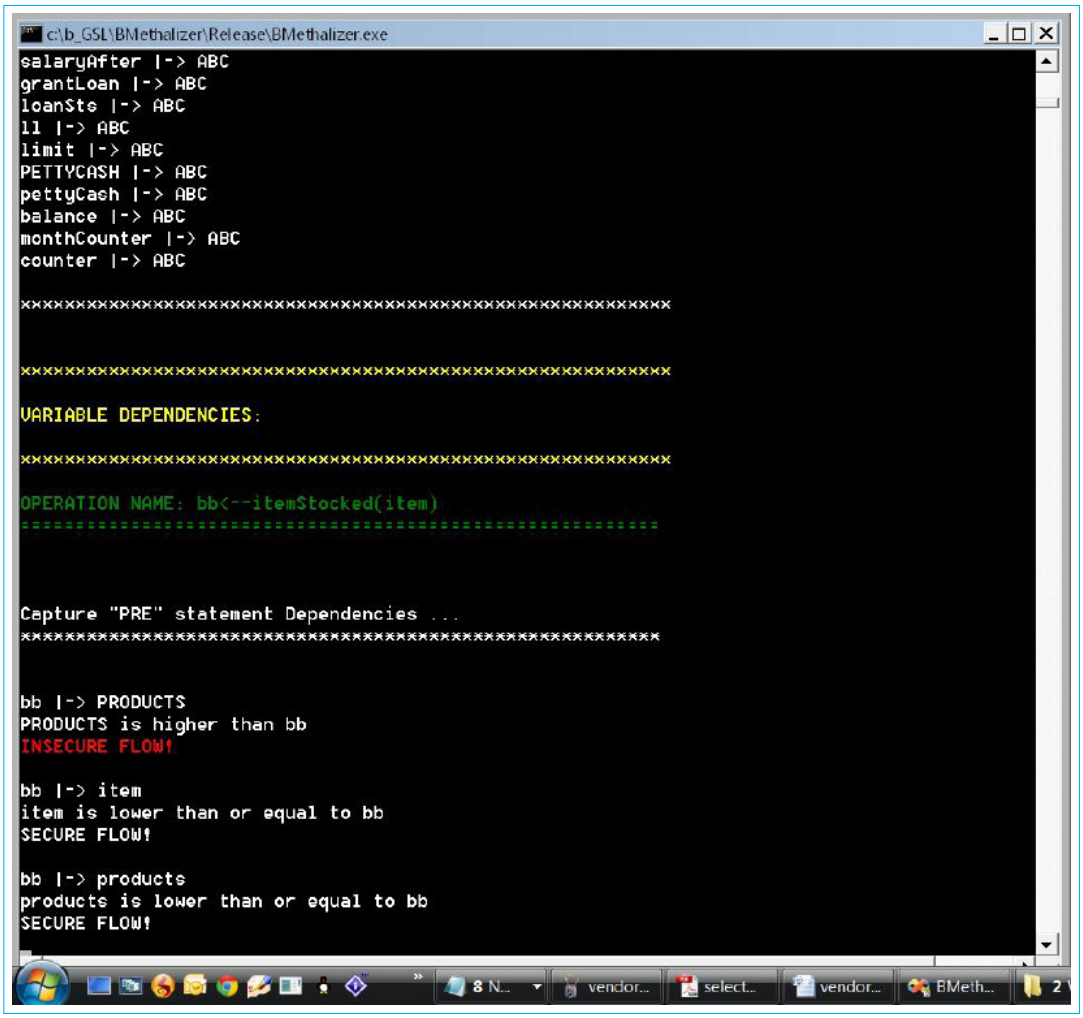


Figure 48: *Vendors : Flow Analysis*

Flow Analyser Screenshots: Refinements

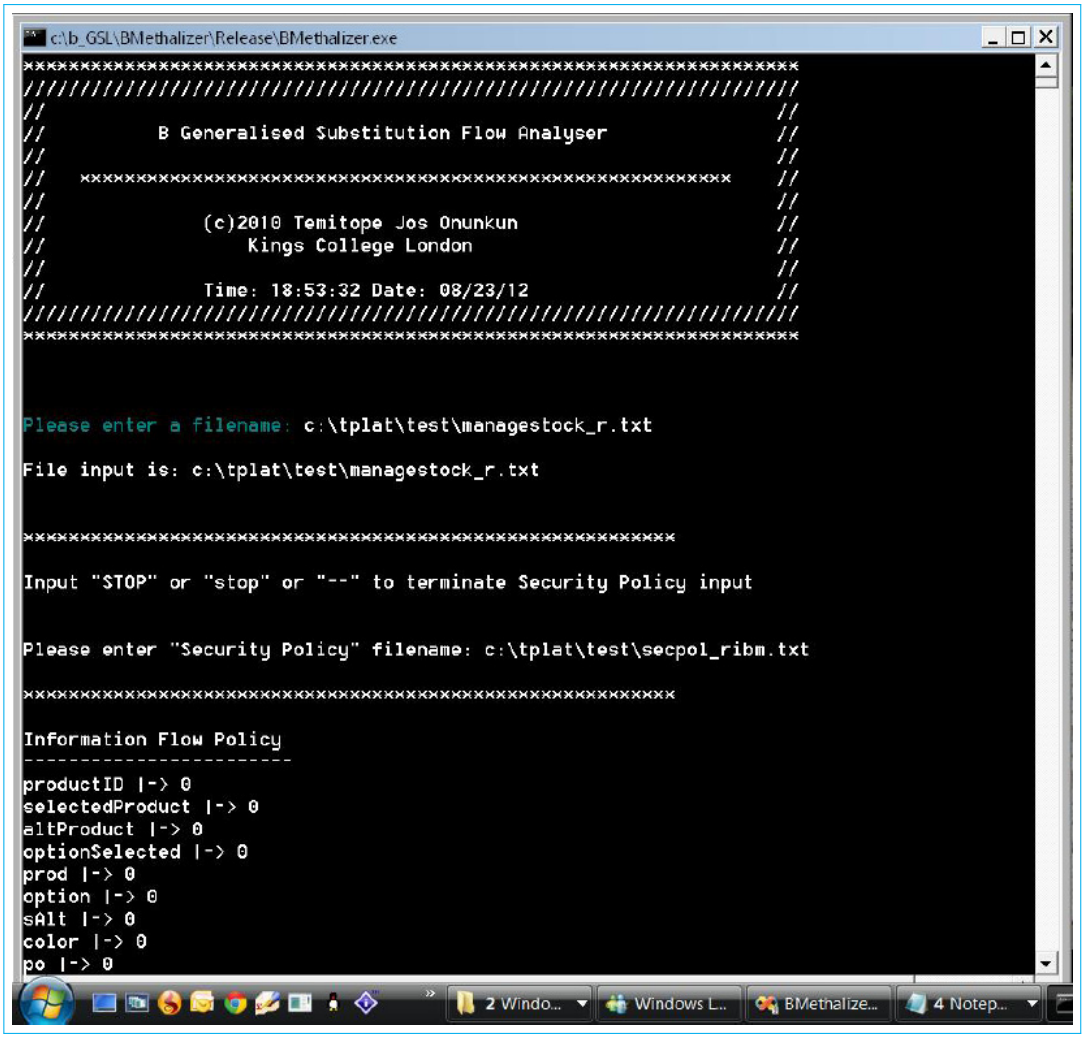


Figure 49: *ManageStock_r : Input File*

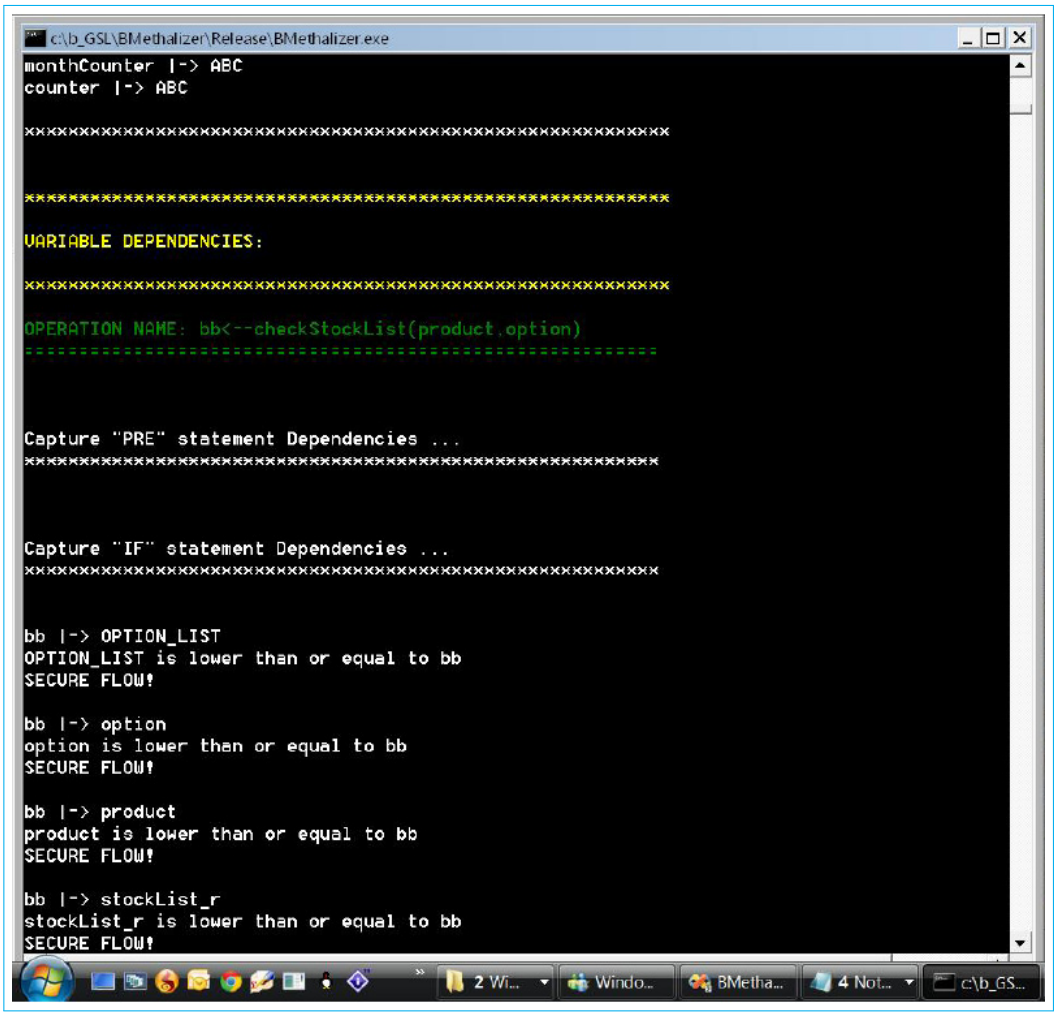


Figure 50: *ManageStock_r : Flow Analysis*

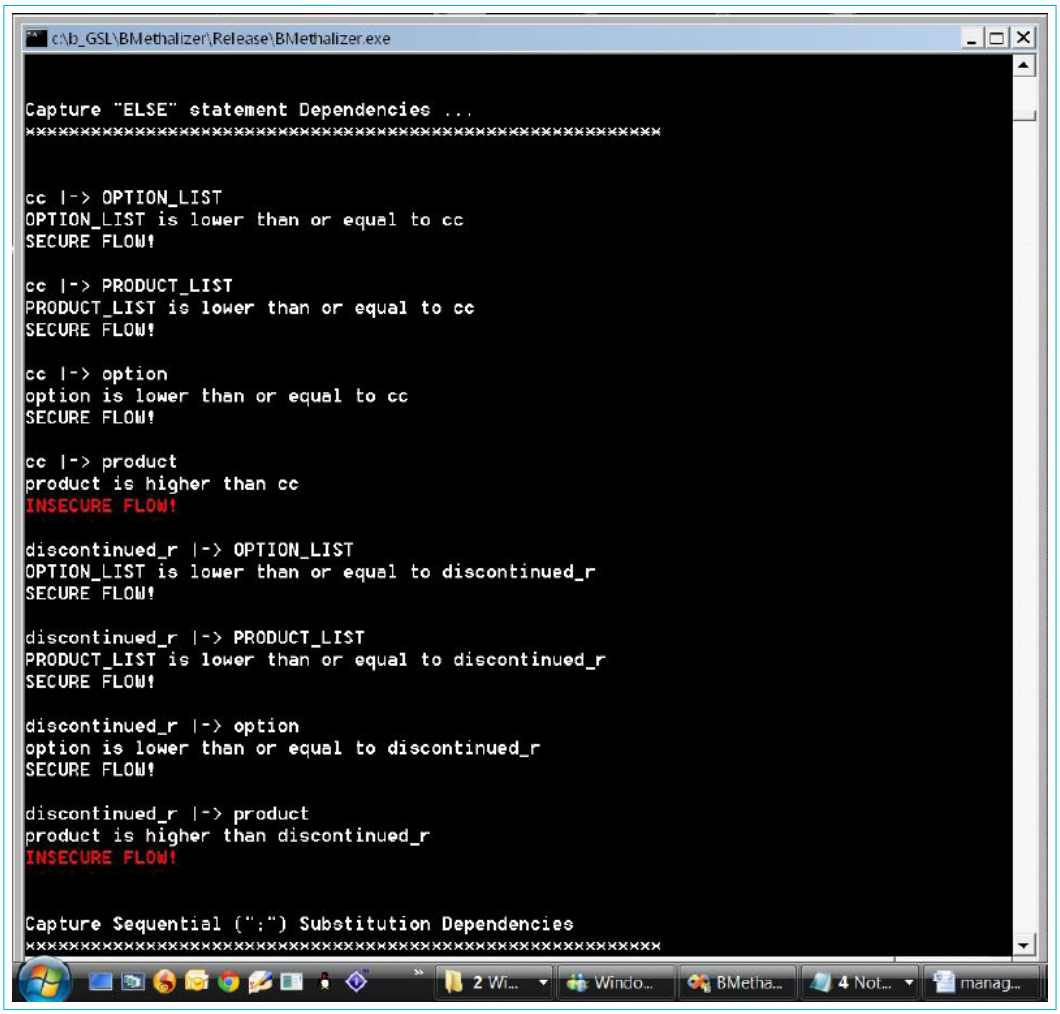


Figure 51: *ManageStock_r : Flow Analysis*

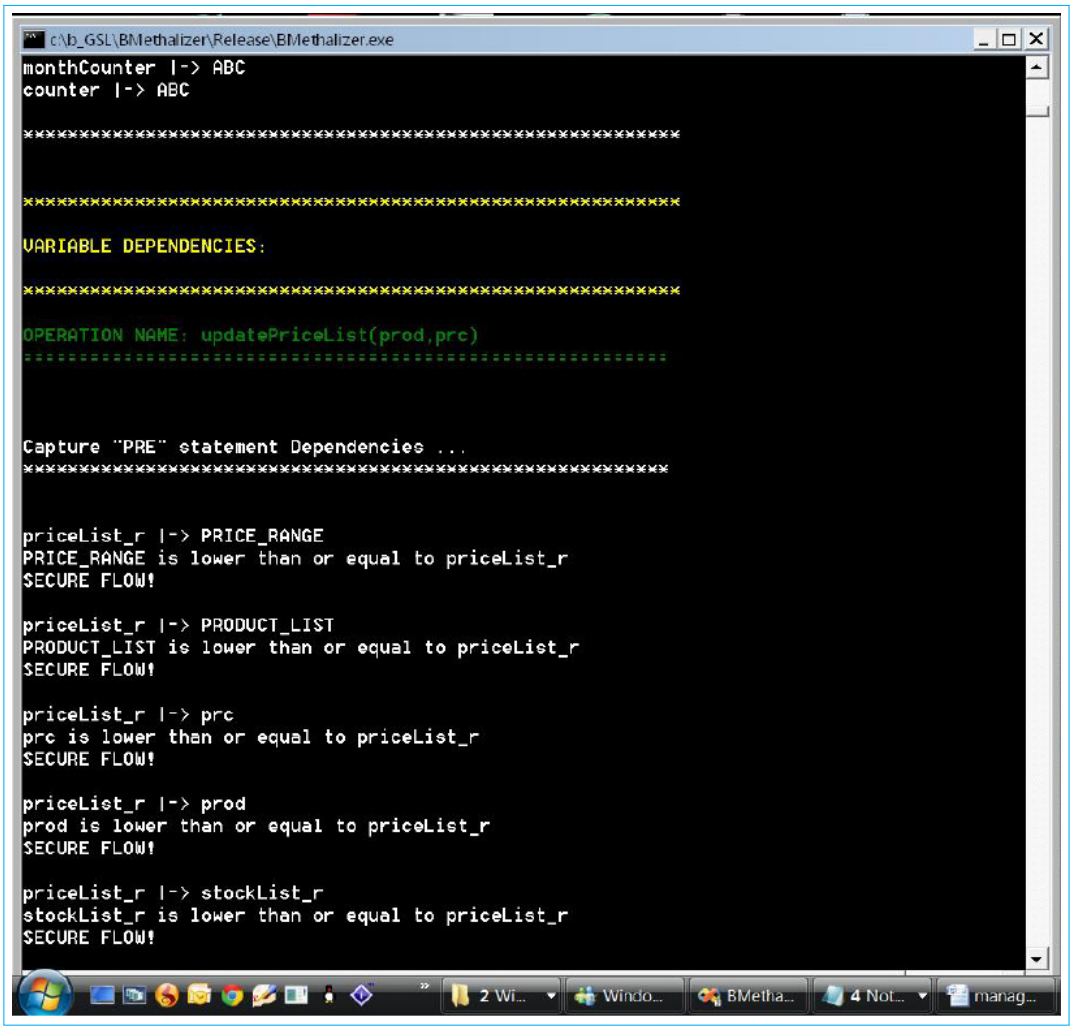


Figure 52: *ManageStock_r : Flow Analysis*

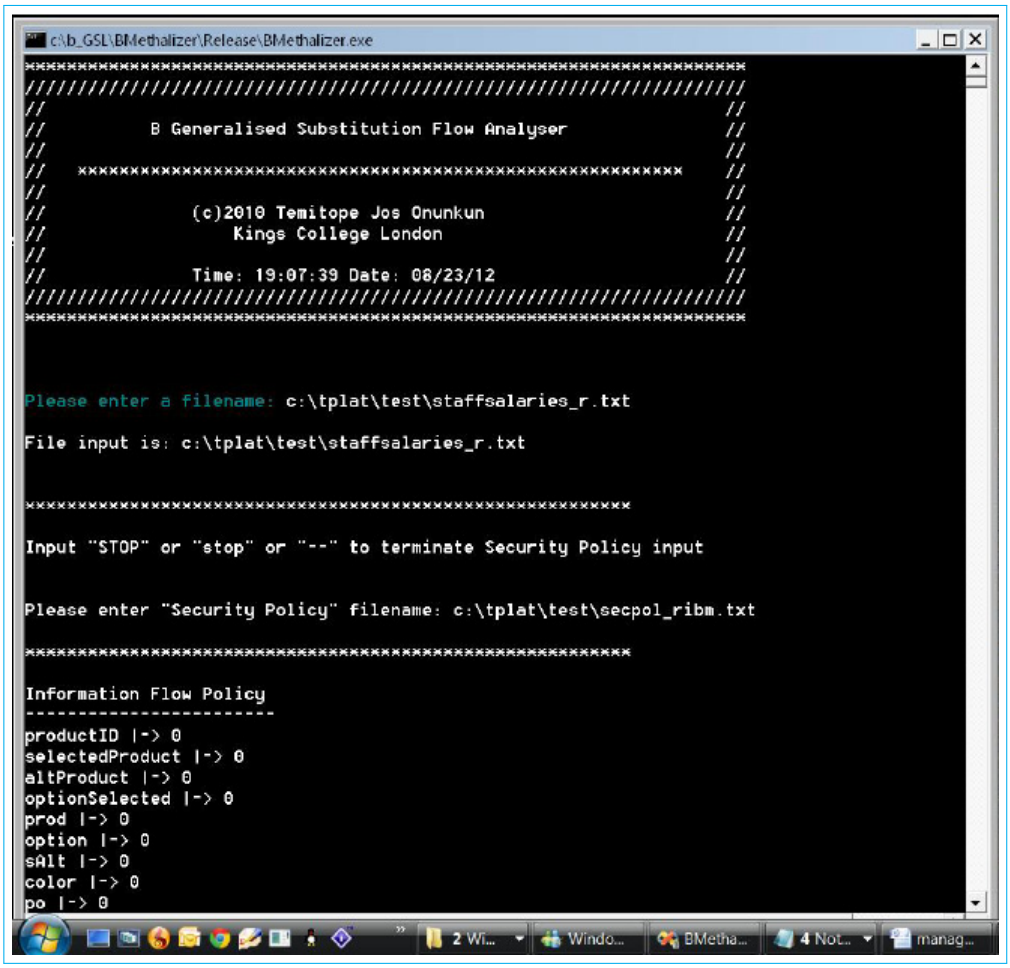


Figure 53: $StaffSalaries\_r$: *Input File*

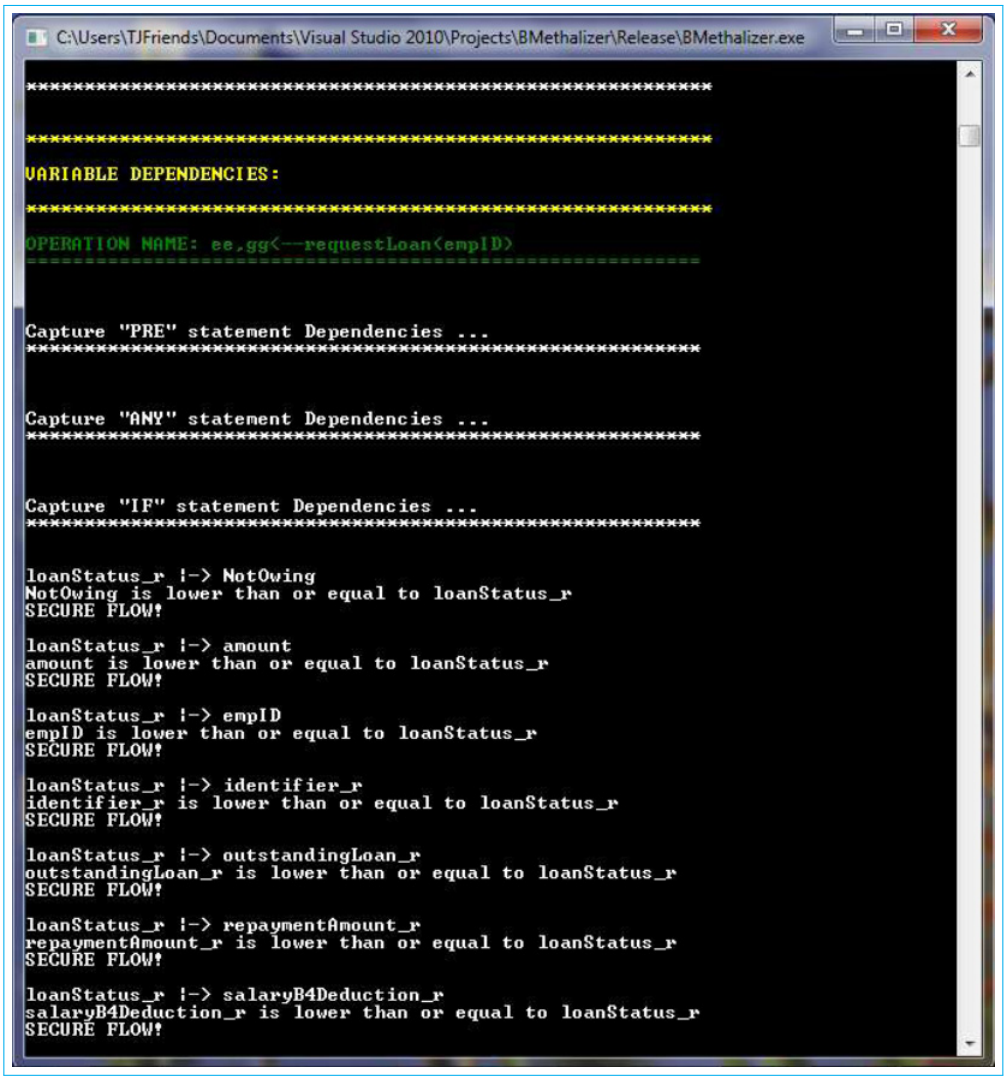


Figure 54: $StaffSalaries\_r: Flow\ Analysis$

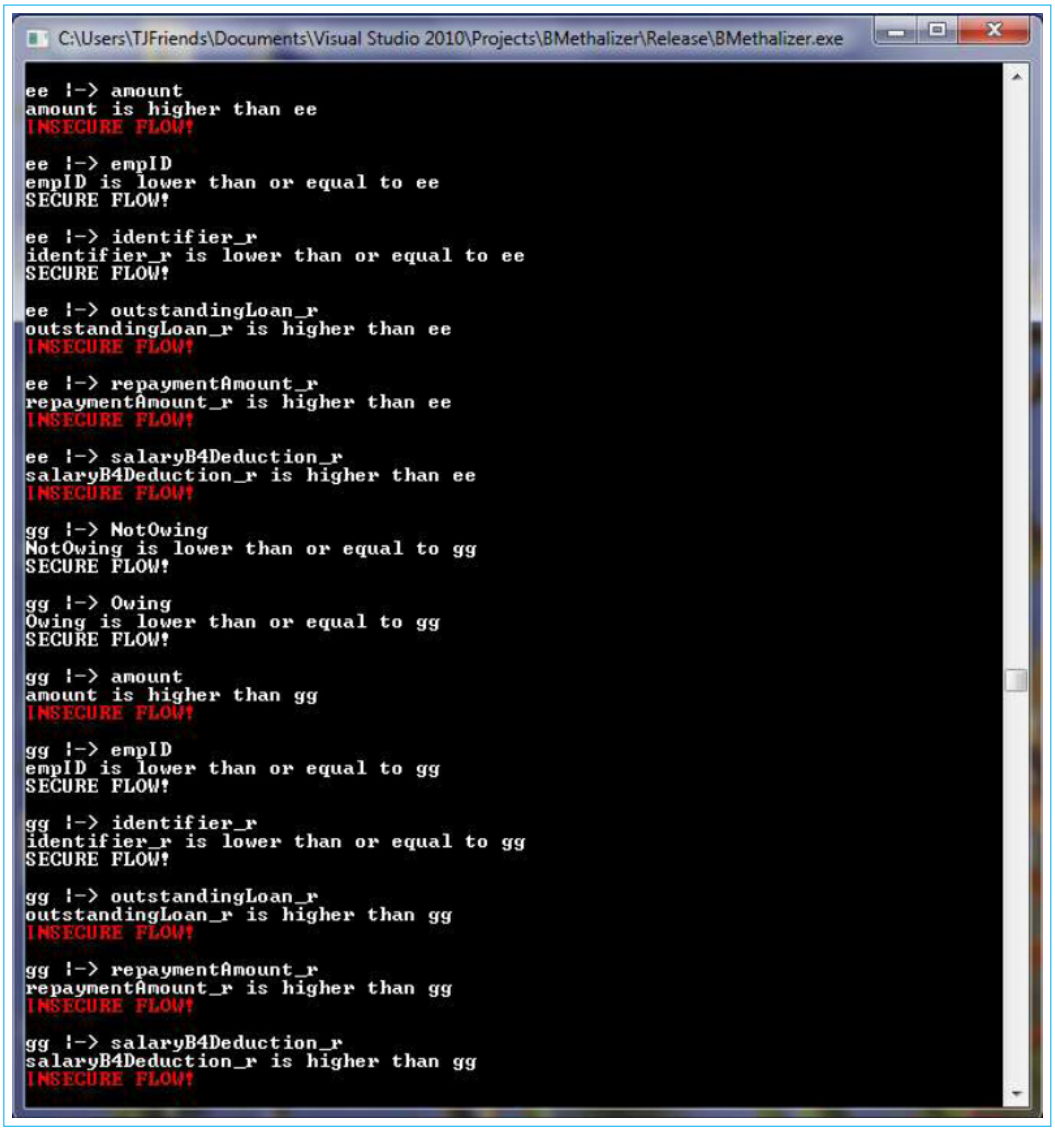


Figure 55: $StaffSalaries\_r: \ Flow \ Analysis$

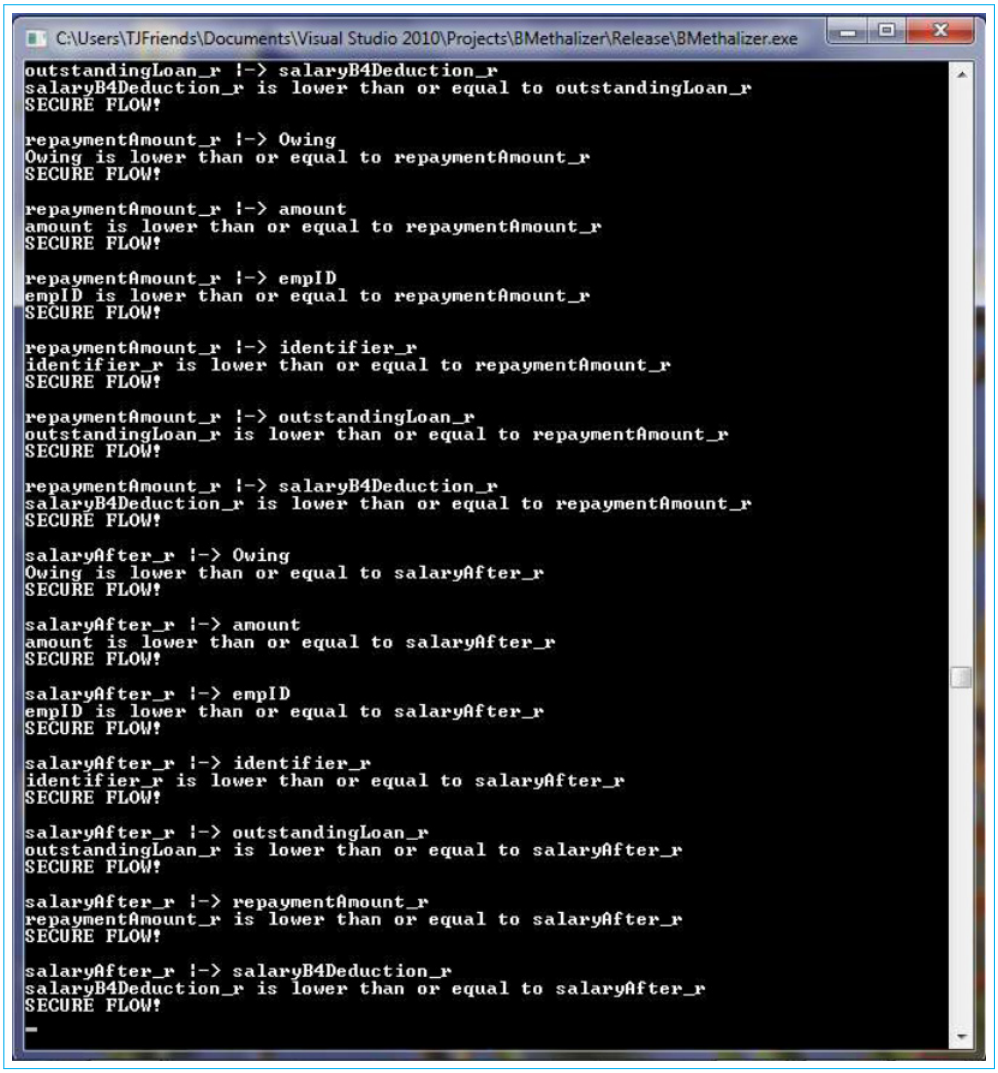


Figure 56: $StaffSalaries\_r: \ Flow \ Analysis$

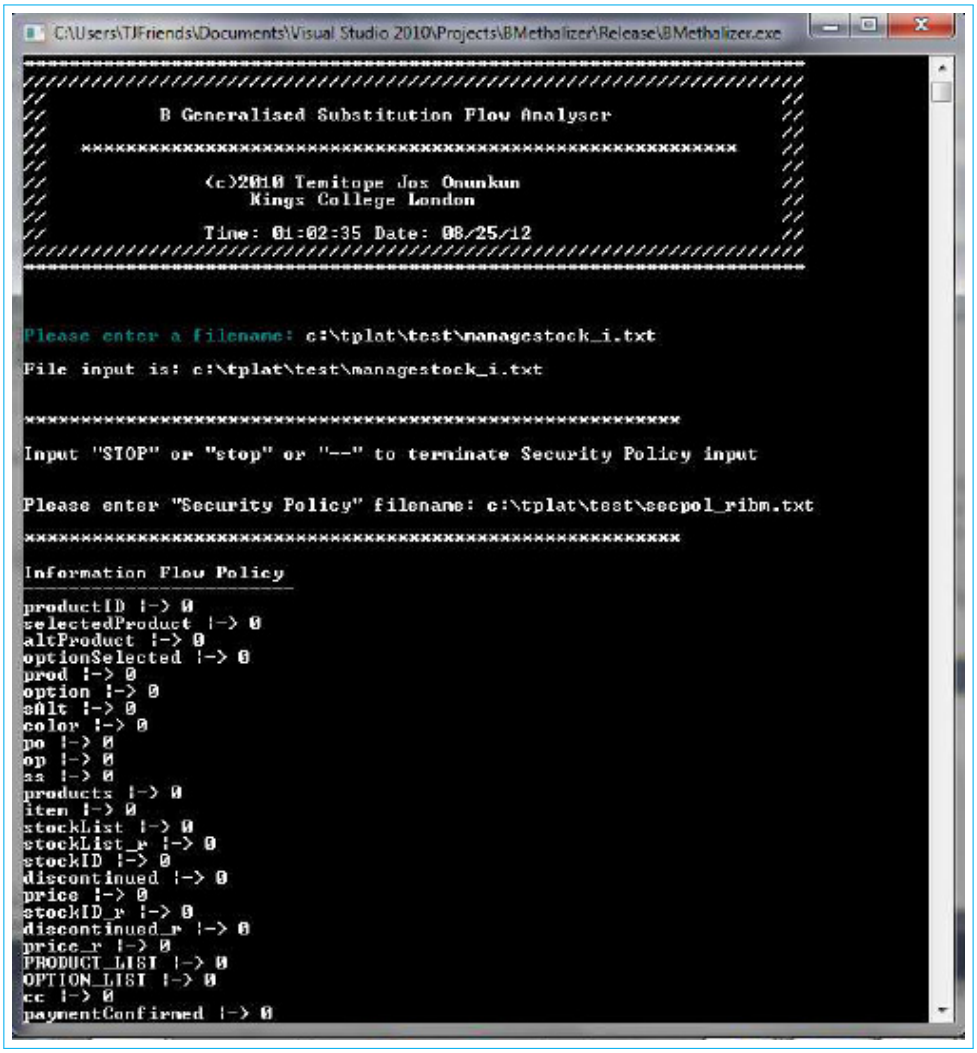


Figure 57: *ManageStock_i* : *Input File*

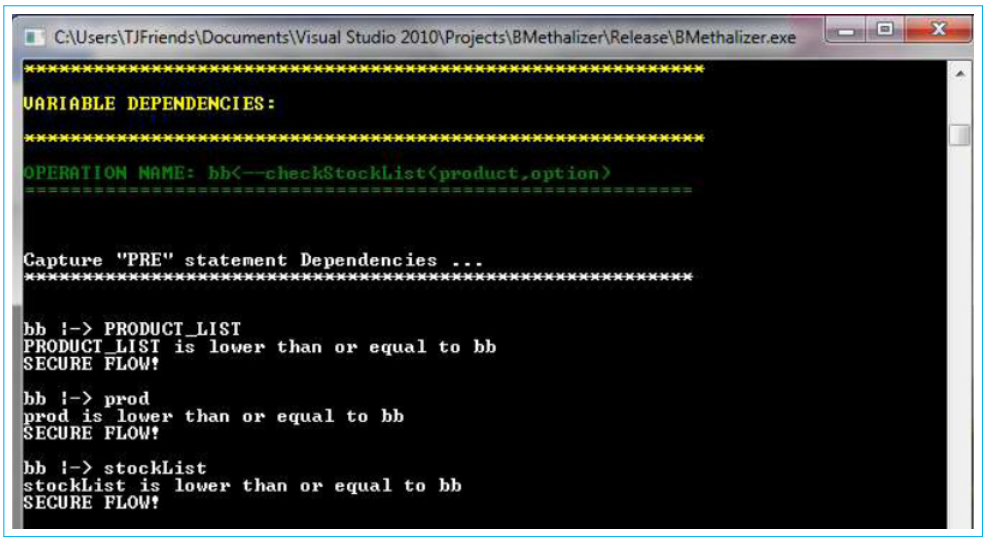


Figure 58: *ManageStock_i : Flow Analysis*

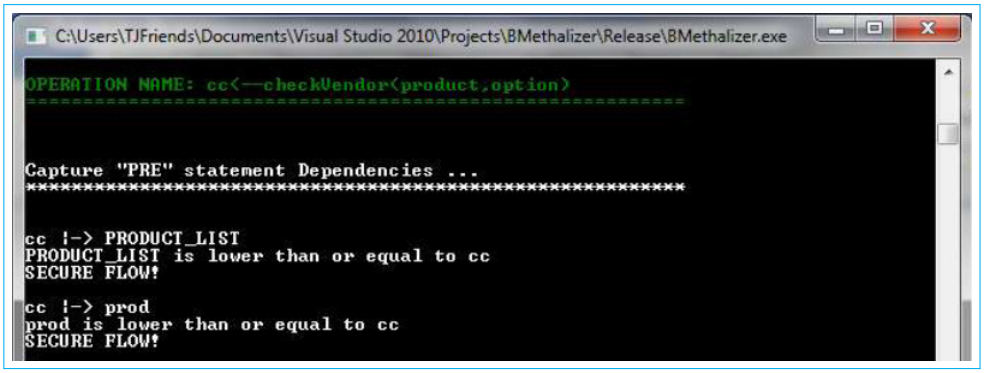


Figure 59: *ManageStock_i : Flow Analysis*

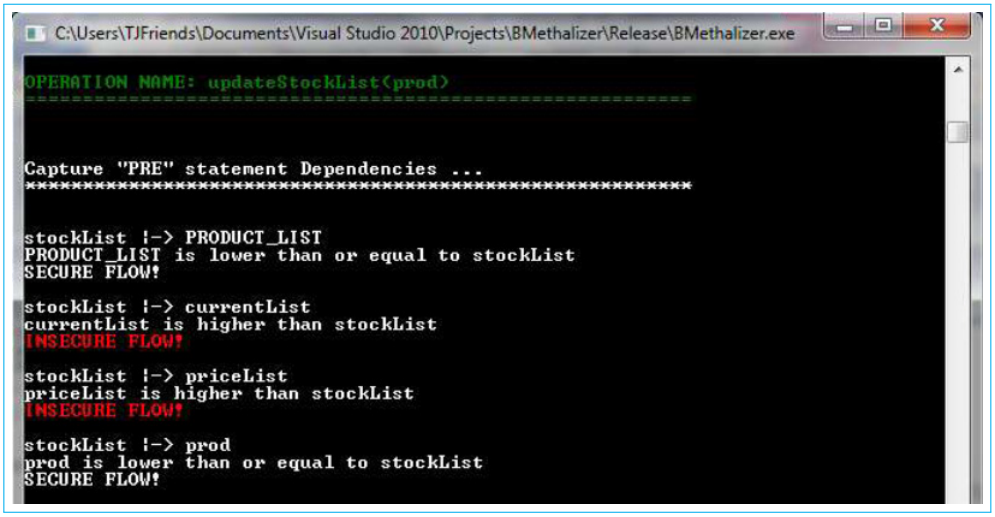


Figure 60: *ManageStock_i : Flow Analysis*

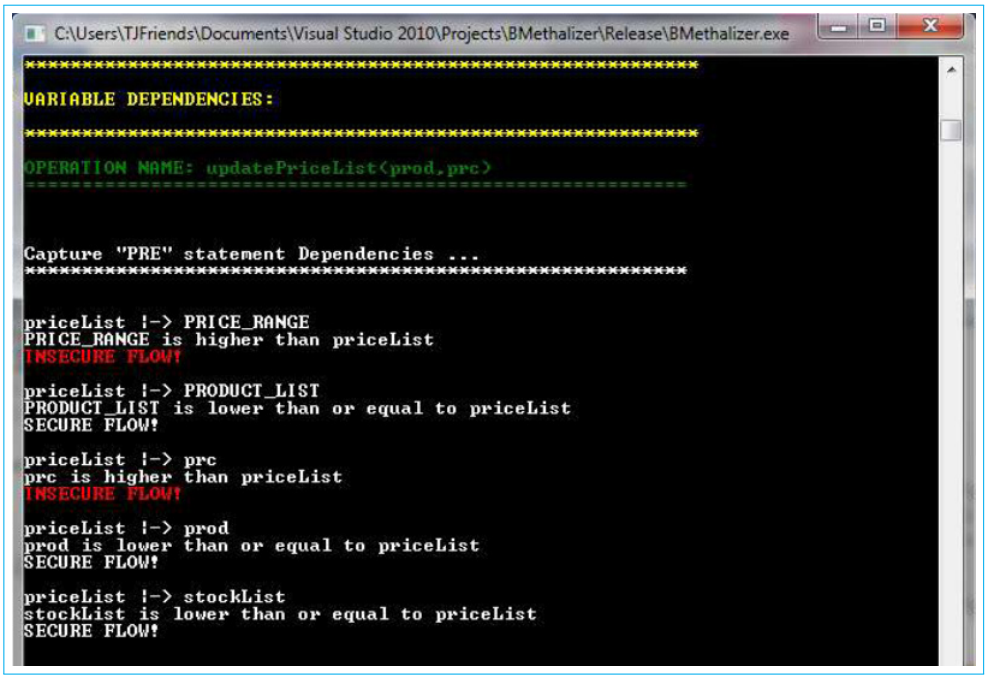


Figure 61: *ManageStock_i : Flow Analysis*

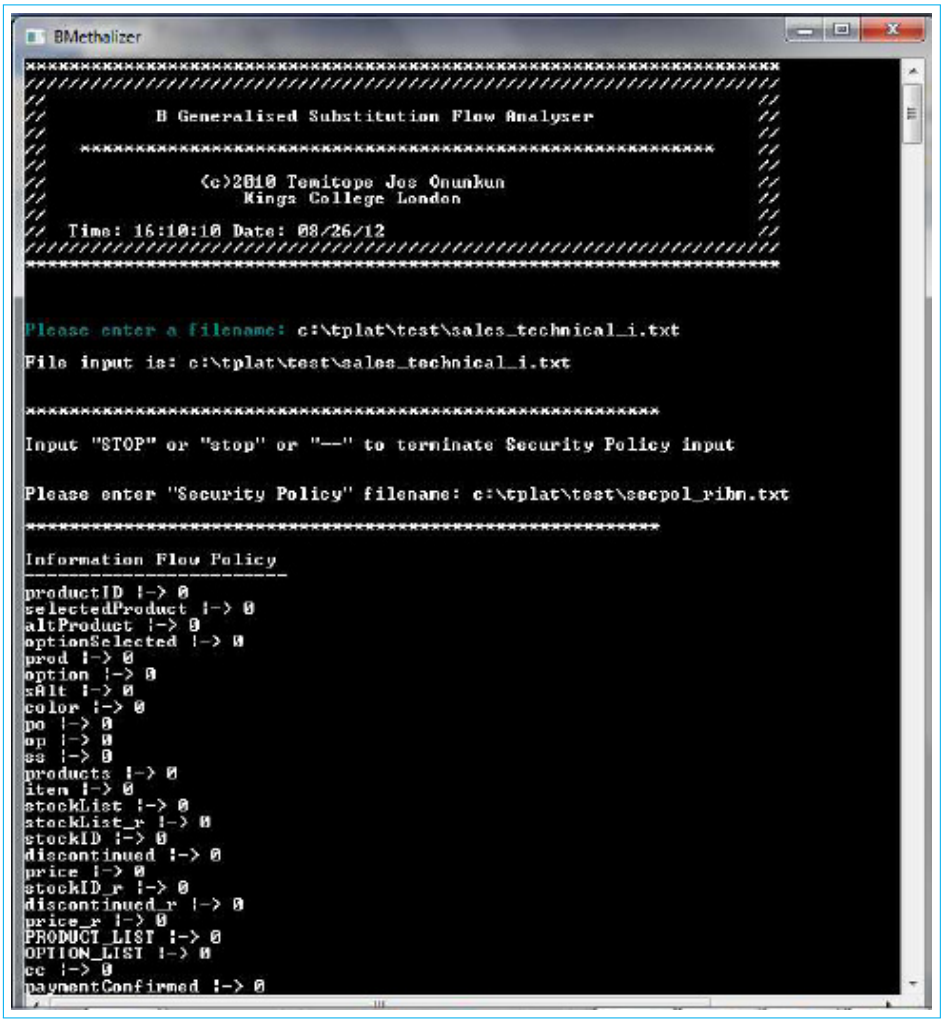


Figure 62: *Sales_Technical_i* : *Input File*

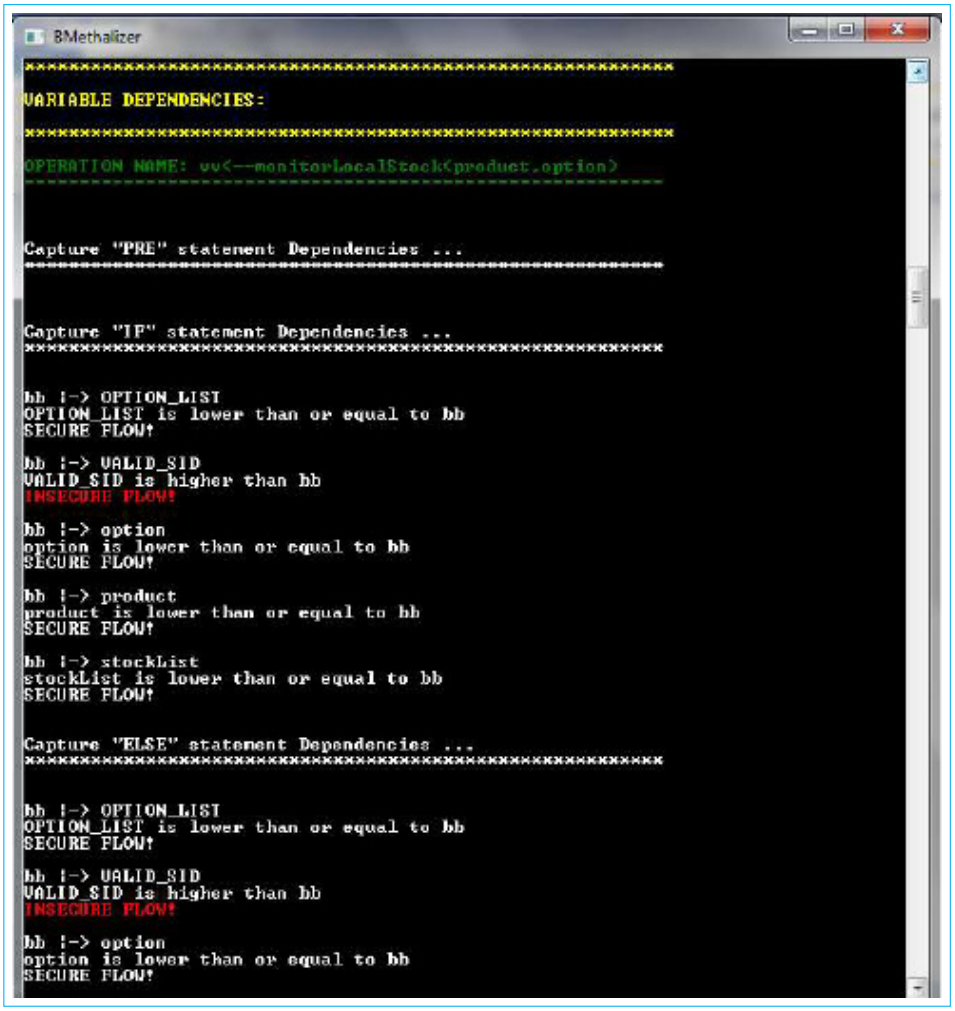


Figure 63: *Sales_Technical_i : Flow Analysis*

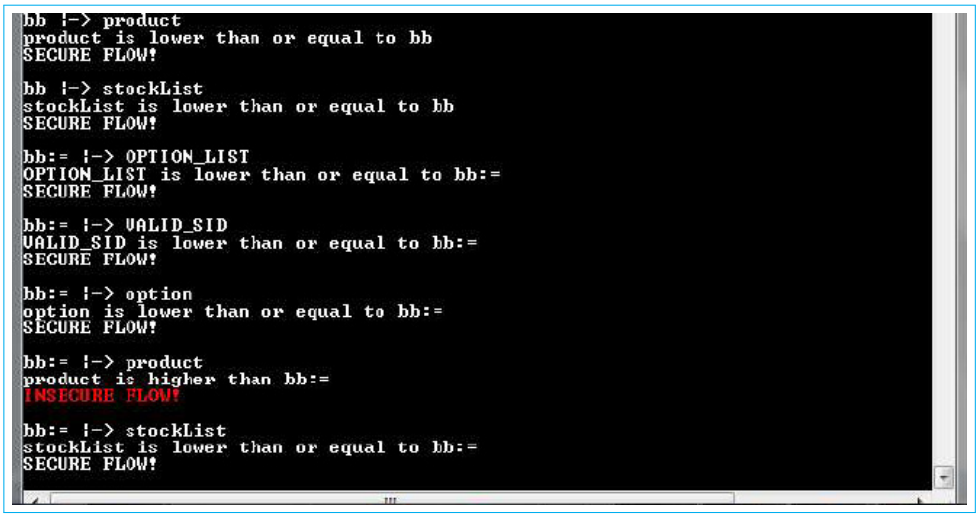


Figure 64: *Sales_Technical_i : Flow Analysis*

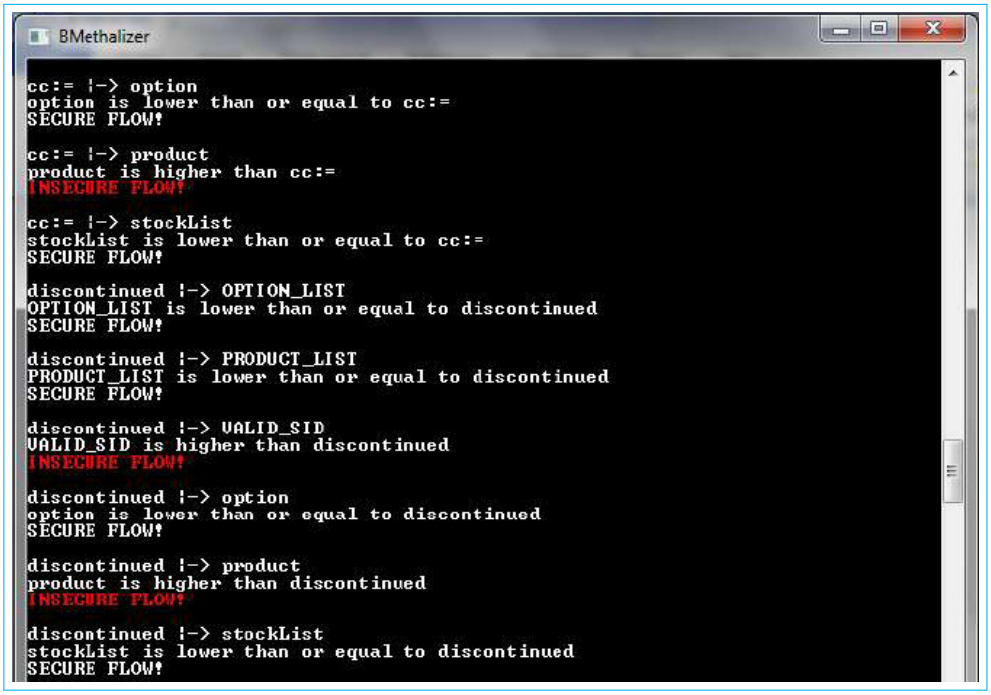


Figure 65: *Sales_Technical_i : Flow Analysis*

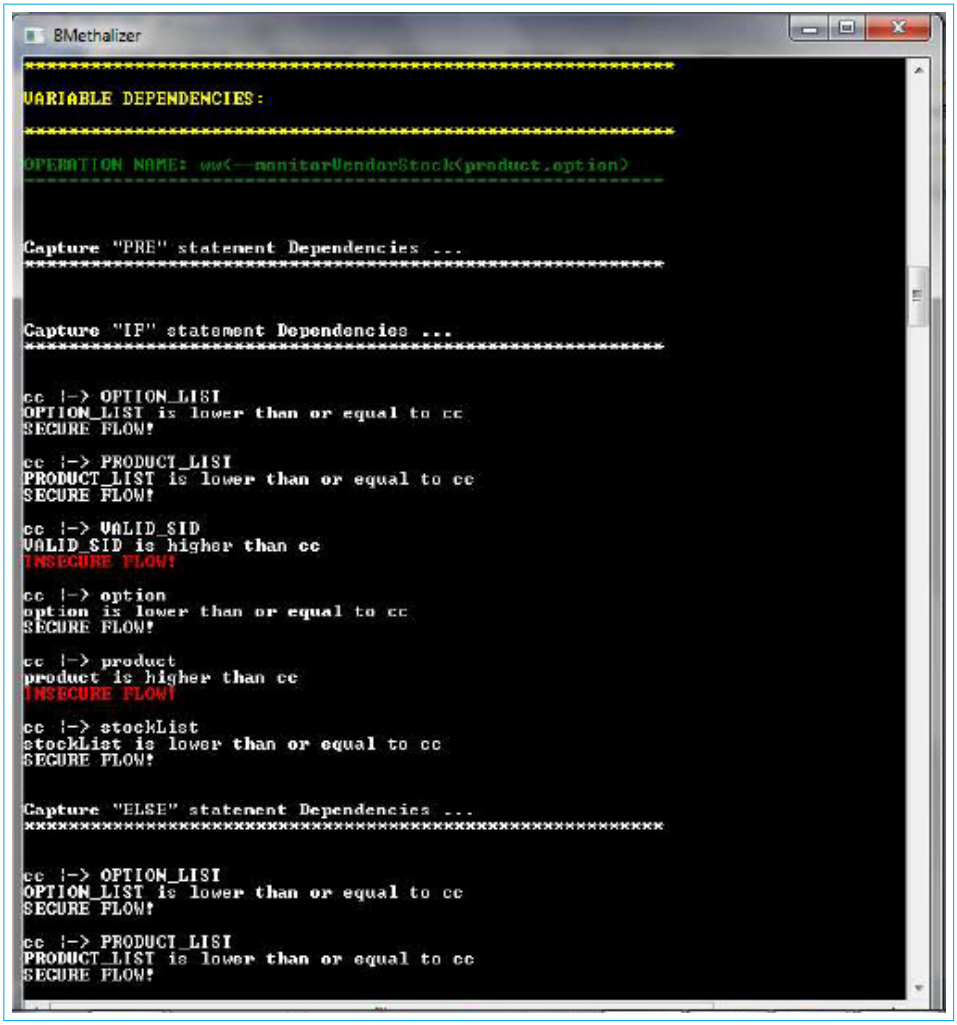


Figure 66: *Sales_Technical_i : Flow Analysis*

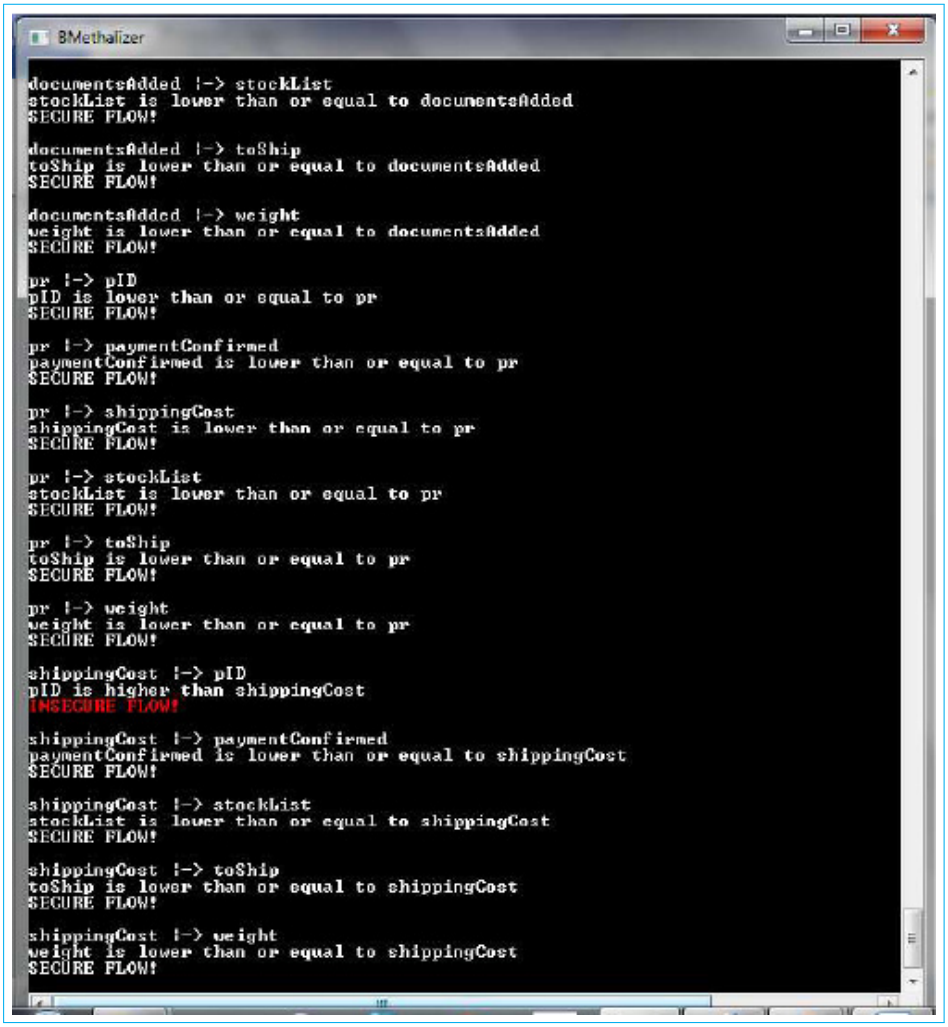


Figure 67: *Sales_Technical_i : Flow Analysis*